Luiz Eduardo Borges

Luiz Eduardo Borges

Python para desenvolvedores

1º EDIÇÃO

RIO DE JANEIRO
EDIÇÃO DO AUTOR
2009

Sumário

Parte I

- Prefácio.
- Introdução.
- Sintaxe.
- Tipos.
- Funções.
- Documentação.
- Exercícios I.

Prefácio

As linguagens dinâmicas eram vistas no passado apenas como linguagens *script*, usadas para automatizar pequenas tarefas, porém, com o passar do tempo, elas cresceram, amadureceram e conquistaram seu espaço no mercado, a ponto de chamar a atenção dos grandes fornecedores de tecnologia.

Vários fatores contribuíram para esta mudança, tais como a internet, o software de código aberto e as metodologias ágeis de desenvolvimento.

A internet viabilizou o compartilhamento de informações de uma forma sem precedentes na história, que tornou possível o crescimento do software de código aberto. As linguagens dinâmicas geralmente são código aberto e compartilham as mesmas funcionalidades e em alguns casos, os mesmos objetivos.

A produtividade e expressividade das linguagens dinâmicas se encaixam perfeitamente com as metodologias ágeis, que nasceram do desenvolvimento de software de código aberto e defendem um enfoque mais pragmático no processo de criação e manutenção de software do que as metodologias mais tradicionais.

Entre as linguagens dinâmicas, o Python se destaca como uma das mais populares e poderosas. Existe uma comunidade movimentada de usuários da linguagem no mundo, o que se reflete em listas de discussão ativas e muitas ferramentas disponíveis em código aberto.

Aprender uma nova linguagem de programação significa aprender a pensar de outra forma. E aprender uma linguagem dinâmica representa uma mudança de paradigma ainda mais forte para aquelas pessoas que passaram anos desenvolvendo em linguagens estáticas.

Introdução

Python[1] é uma linguagem de altíssimo nível (em inglês, *Very High Level Language*) orientada a objetos, de tipagem dinâmica e forte, interpretada e interativa.

Características

O Python possui uma sintaxe clara e concisa, que favorece a legibilidade do código fonte, tornando a linguagem mais produtiva.

A linguagem inclui diversas estruturas de alto nível (listas, tuplas, dicionários, data / hora, complexos e outras) e uma vasta coleção de módulos prontos para uso, além de *frameworks* de terceiros que podem ser adicionados. Também possui recursos encontrados em outras linguagens modernas, tais como: geradores, introspecção, persistência, metaclasses e unidades de teste. Multiparadigma, a linguagem suporta programação modular e funcional, além da orientação a objetos. Mesmo os tipos básicos no Python são objetos.
A linguagem é interpretada através de *bytecode* pela máquina virtual Python, tornando o código portável. Com isso é possível compilar aplicações em uma plataforma e rodar em outras ou executar direto do código fonte.

Python é um software de código aberto (com licença compatível com a *General Public License* (GPL), porém menos restritiva, permitindo que o Python seja incorporados em produtos proprietários) e a especificação da linguagem é mantida pela *Python Software Foundation*[2] (PSF).

Python é muito utilizado como linguagem *script* em vários softwares, permitindo automatizar tarefas e adicionar novas funcionalidades, entre eles: BrOffice.org, PostgreSQL, Blender e GIMP. Também é possível integrar o Python a outras linguagens, como a Linguagem C. Em termos gerais, o Python apresenta muitas similaridades com outras linguagens dinâmicas, como Perl e Ruby.

Histórico

Python foi criada em 1990 por Guido van Rossum, no Instituto Nacional de Pesquisa para Matemática e Ciência da Computação da Holanda (CWI) e tinha como foco original usuários como físicos e engenheiros. O Python foi concebido a partir de outra linguagem existente na época, chamada ABC.

Hoje, a linguagem é bem aceita na industria por empresas de alta tecnologia, tais como:

1 Página oficial: http://www.python.org/.
2 Endereço na internet da PSF: http://www.python.org/psf/.

- Google (aplicações *Web*).
- Yahoo (aplicações *Web*).
- Microsoft (IronPython: Python para .NET).
- Nokia (disponível para as linhas recentes de celulares e PDAs).
- Disney (animações 3D).

Exemplo

Exemplo de programa em Python:

```
# Exemplo de programa em Python

# Uma lista de instrumentos musicais
instrumentos = ['Baixo', 'Bateria', 'Guitarra']

# Para cada nome na lista de instrumentos
for instrumento in instrumentos:

    # mostre o nome do instrumento musical
    print instrumento
```

Saída:

```
Baixo
Bateria
Guitarra
```

No exemplo, “instrumentos” é uma lista contendo itens “Baixo”, “Bateria” e “Guitarra”. Já “instrumento” é um nome que corresponde a cada um dos itens da lista, conforme o laço é executado.

Versões

A implementação oficial do Python é mantida pela PSF e escrita em C, e por isso é também conhecida como CPython. A versão estável mais recente está disponível para *download* no endereço:

http://www.python.org/download/

Para Windows, basta executar o instalador. Para outras plataformas, geralmente o Python já faz parte do sistema, porém em alguns casos pode ser necessário compilar e instalar a partir dos arquivos fonte.

Existem também implementações de Python para .NET (IronPython) e JVM (Jython).

Tipagem dinâmica

Python utiliza tipagem dinâmica, o que significa que o tipo de uma variável é inferido pelo interpretador em tempo de execução (isto é conhecido como *Duck Typing*). No momento em que uma variável é criada através de atribuição, o interpretador define um tipo para a variável, com as operações que podem ser aplicadas.

O Python tem tipagem forte, ou seja, ele verifica se as operações são válidas e não faz coerções automáticas entre tipos incompatíveis[3]. Para realizar a operação entre tipos não compatíveis, é necessário converter explicitamente o tipo da variável ou variáveis antes da operação.

Bytecode

O código fonte é traduzido pelo interpretador para o formato *bytecode*, que é multiplataforma e pode ser executado e distribuído sem fonte original.

Por padrão, o interpretador compila os fontes e armazena o *bytecode* em disco, para que a próxima vez que o executar, não precise compilar novamente o programa, reduzindo o tempo de carga na execução. Se os fontes forem alterados, o interpretador se encarregará de regerar o *bytecode* automaticamente, mesmo utilizando o *shell* interativo. Quando um programa ou um módulo é evocado, o interpretador realiza a análise do código, converte para símbolos, compila (se não houver *bytecode* atualizado em disco) e executa na máquina virtual Python.

O *bytecode* é armazenado em arquivos com extensão *".pyc"* (*bytecode* normal) ou *".pyo"* (*bytecode* otimizado).

O *bytecode* também pode ser empacotado junto com o interpretador em um executável, para facilitar a distribuição da aplicação.

Modo interativo

O interpretador Python pode ser usado de forma interativa, na qual as linhas de código são digitadas em um *prompt* (linha de comando) semelhante ao *shell* do sistema operacional.

Para evocar o modo interativo basta rodar o Python (se ele estiver no *path*):

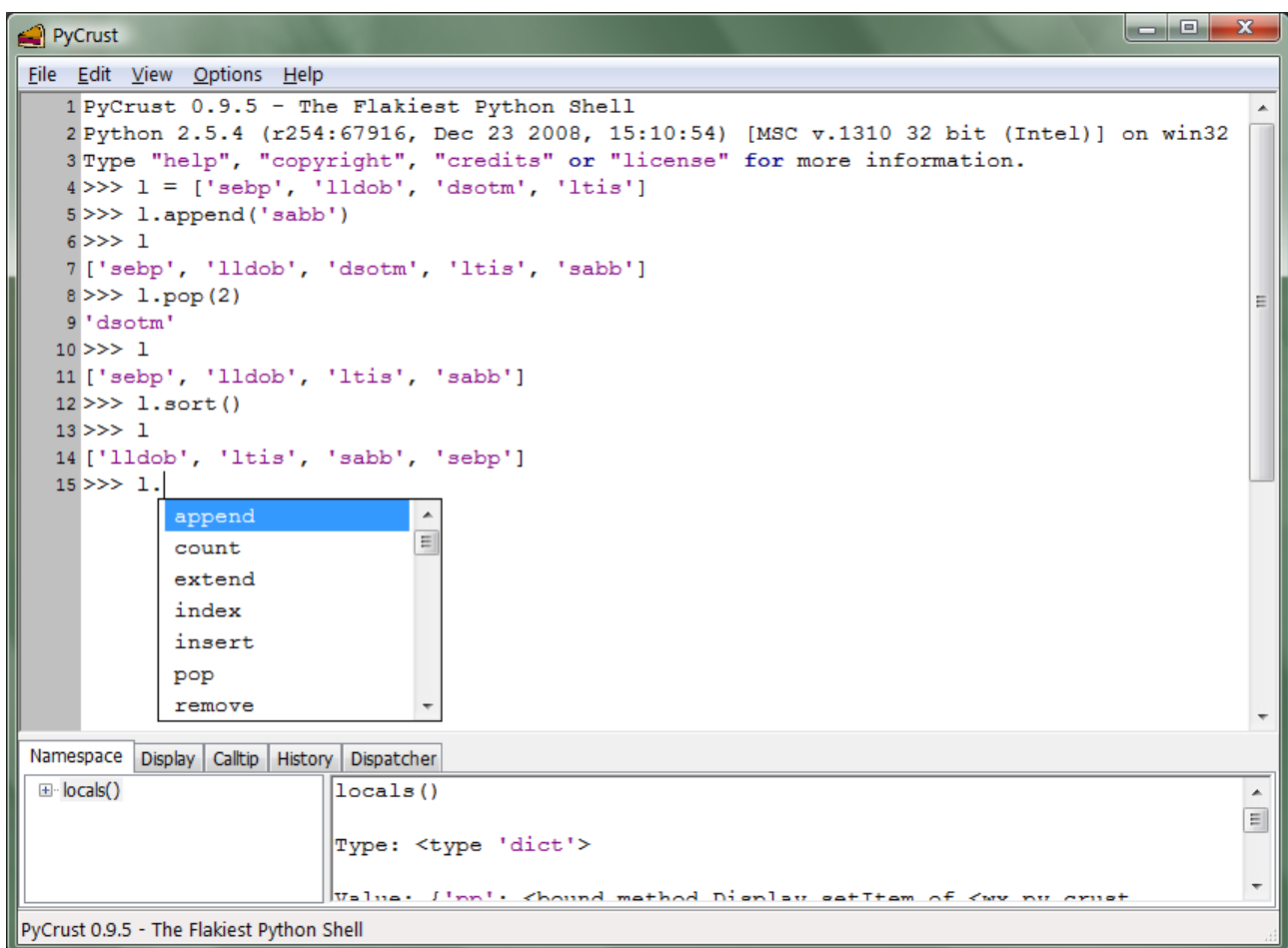
---

3 Em Python, coerções são realizadas automaticamente apenas entre tipos que são claramente relacionados, como inteiro e inteiro longo.

```
python
```

Ele estará pronto para receber comandos surgir o *prompt* ">>>" na tela.

O modo interativo é uma característica diferencial da linguagem, pois é possível testar e modificar o código de um programa antes da inclusão do código nos programas, por exemplo.

Exemplo do Python sendo usado de forma interativa (com o *shell* PyCrust[4]):

Os arquivos fonte normalmente são identificados pela extensão "*.py*" e podem ser executados diretamente pelo interpretador:

```
python apl.py
```

4 PyCrust é uma ferramenta que faz parte do projeto wxPython (http://www.wxpython.org/).

Assim o programa “apl.py” será executado.

Ferramentas

Existem muitas ferramentas de desenvolvimento para Python, como IDEs, editores e *shells* (que aproveitam da capacidade interativa do Python).

Integrated Development Environments (IDEs) são pacotes de software integram várias ferramentas de desenvolvimento em um ambiente integrado, com o objetivo de aumentar a produtividade do desenvolvedor.

Entre as IDEs que suportam Python, encontram-se:

- SPE (Stani's Python Editor).
- Eric.
- Open Komodo.
- PyDev (plugin para a IDE Eclipse).

Existem também editores de texto especializados em código de programação, que possuem funcionalidades como colorização de sintaxe, exportação para outros formatos e conversão de codificação de texto.

Esses editores suportam diversas linguagens de programação, dentre elas o Python:

- SciTE.
- Notepad++.

Shells são ambientes interativos para execução de comandos, muito úteis para testar pequenas porções de código e para atividades como *data crunching* (extrair informações de interessa de massas de dados ou traduzir dados de um formato para outro).

Além do próprio *Shell* padrão do Python, existem os outros disponíveis:

- PyCrust (gráfico).
- Ipython (texto).

Empacotadores são utilitários que empacotam o bytecode com o interpretador e outras dependências, em um ou mais executáveis e arquivos compactados, permitindo que o aplicativo rode em máquinas sem Python instalado.

Entre empacotadores feitos para Python, estão disponíveis:

- Py2exe (apenas para Windows).
- cx_Freeze (portável).

Frameworks são coleções de componentes de software (bibliotecas, utilitários e outros) que foram projetados para serem utilizados por outros sistemas.

Alguns *frameworks* disponíveis mais conhecidos:

- Web: Django, TurboGears e Zope.
- Interface gráfica: wxPython, PyGTK e PyQT.
- Processamento cientifico: NumPy.
- Processamento de imagens: PIL.
- 2D: Matplotlib.
- 3D: Visual Python, PyOpenGL e Python Ogre.
- Mapeamento objeto-relacional: SQLAlchemy e SQLObject.

Cultura

O nome Python foi tirado por Guido van Rossum do programa da TV britânica *Monty Python Flying Circus*, e existem várias referências na documentação da linguagem ao programa, como, por exemplo, o repositório de pacotes oficial do Python se chamava *Cheese Shop*, que era o nome de um dos quadros do programa. Atualmente, o nome do repositório é *Python Package Index*[5] (PYPI).

A comunidade de usuários de Python criou algumas expressões para se referir aos assuntos relacionados à linguagem. Neste jargão, o termo *Pythonic* é usado para indicar que algo é compatível com as premissas de projeto do Python, e *Unpythonic* significa o oposto. Já o usuário da linguagem é chamado de *Pythonist*.

As metas do projeto foram resumidas por Tim Peters em um texto chamado *Zen of Python,* que está disponível no próprio Python através do comando:

```
import this
```

O texto enfatiza a postura pragmática do *Benevolent Dictator for Life* (BDFL), como Guido é conhecido na comunidade Python.

Propostas para melhoria da linguagem são chamadas de PEPs (*Python Enhancement Proposals*), que também servem de referência para novos recursos a serem implementados na linguagem.

Além do site oficial, outras boas fontes de informação sobre a linguagem são:

5 Endereço: http://pypi.python.org/pypi.

PythonBrasil[6], o *site* da comunidade Python no Brasil, com bastante informação em português, e Python Cookbook[7], site que armazena "receitas": pequenas porções de código para realizar tarefas específicas.

6 Endereço: http://www.pythonbrasil.com.br/.
7 Endereço: http://aspn.activestate.com/ASPN/Python/Cookbook/.

Sintaxe

Um programa feito em Python é constituído de linhas, que podem continuar nas linhas seguintes, pelo uso do caractere de barra invertida (\) ao final da linha ou parênteses, colchetes ou chaves, em expressões que utilizam tais caracteres.

O caractere # marca o inicio de comentário. Qualquer texto depois do # será ignorado até o fim da linha , com exceção dos comentários funcionais.

Comentários funcionais geralmente são usados para:

- alterar a codificação do arquivo fonte do programa acrescentando um comentário com o texto “#-*- coding: <encoding> -*#-” no inicio do arquivo, aonde <encoding> é a codificação do arquivo (geralmente *latin1* ou *utf-8*). Alterar a codificação é necessário para suportar caracteres que não fazem parte da linguagem inglesa, no código fonte do programa.
- definir o interpretador que será utilizado para rodar o programa em sistemas UNIX, através de um comentário começando com “#!” no inicio do arquivo, que indica o caminho para o interpretador (geralmente a linha de comentário será algo como “#!/usr/bin/env python”).

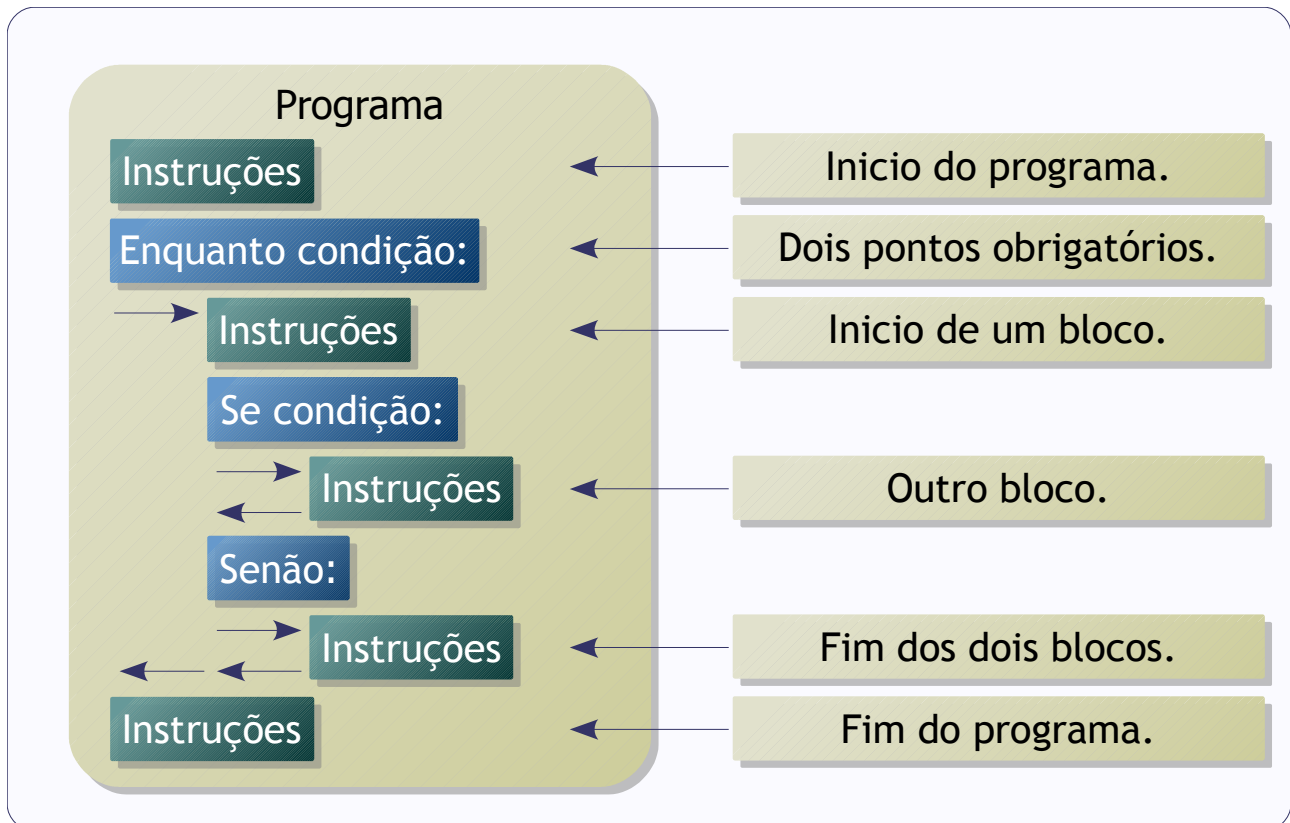
Exemplo de comentários funcionais:

```
#!/usr/bin/env python
# -*- coding: latin1 -*-

# Uma linha de código que mostra o resultado de 7 vezes 3
print 7 * 3
```

Exemplos de linhas quebradas:

```
# Uma linha quebrada por contra-barra
a = 7 * 3 + \
5 / 2

# Uma lista (quebrada por vírgula)
b = ['a', 'b', 'c',
'd', 'e']

# Uma chamada de função (quebrada por vírgula)
c = range(1,
11)
```

```
# imprime todos na tela
print a, b, c
```

Em Python, os blocos de código são delimitados pelo uso de indentação. A indentação deve ser constante no bloco de código, porém é considerada uma boa prática manter a consistência no projeto todo e evitar a mistura tabulações e espaços[8].

Exemplo:

```
# Para i na lista 234, 654, 378, 798:
for i in [234, 654, 378, 798]:

    # Se o resto dividindo por 3 for igual a zero:
    if i % 3 == 0:

        # Imprime...
        print i, '/ 3 =', i / 3
```

Saída:

8 A recomendação oficial de estilo de codificação (http://www.python.org/dev/peps/pep-0008/) é usar quatro espaços para indentação e esta convenção é amplamente aceita pelos desenvolvedores.

```
234 / 3 = 78
654 / 3 = 218
378 / 3 = 126
798 / 3 = 266
```

O comando *print* coloca espaços entre as expressões que forem recebidas como parâmetro e um caractere de nova linha no final, a não ser que ele receba uma vírgula no fim da lista parâmetros.

O Python é uma linguagem orientada a objeto, sendo assim as variáveis atributos (dados) e métodos (rotinas associadas ao objeto). Tanto os atributos quanto os métodos são acessados usando ponto (.), sendo que os métodos devem ser sempre seguidos de parênteses:

Para mostrar um atributo:

```
print objeto.atributo
```

Para executar um método:

```
objeto.metodo(argumentos)
```

Mesmo um método sem argumentos precisa de parênteses:

```
objeto.metodo()
```

O ponto também é usado para acessar estruturas de módulos que foram importados pelo programa.

Controle de fluxo

É muito comum em um programa que certos conjuntos de instruções sejam executados de forma condicional, em casos como validar entradas de dados, por exemplo.

Sintaxe:

```
if <condição>:
    <bloco de código>
elif <condição>:
    <bloco de código>
elif <condição>:
    <bloco de código>
else:
    <bloco de código>
```

Aonde:

- <condição>: sentença que possa ser avaliada como verdadeira ou falsa.
- <bloco de código>: seqüência de linhas de comando.
- As clausulas *elif* e *else* são opcionais e podem existir vários *elifs* para o mesmo *if*.
- Parênteses só são necessários para evitar ambigüidades.

Exemplo:

```
temp = int(raw_input('Entre com a temperatura: '))

if temp < 0:
    print 'Congelando...'
elif 0 <= temp <= 20:
    print 'Frio'
elif 21 <= temp <= 25:
    print 'Normal'
elif 26 <= temp <= 35:
    print 'Quente'
else:
    print 'Muito quente!'
```

Se o bloco de código for composto de apenas uma linha, ele pode ser escrito após os dois pontos:

```
if temp < 0: print 'Congelando...'
```

A partir da versão 2.5, o Python suporta a expressão:

```
<variável> = <valor 1> if <condição> else <valor 2>
```

Aonde <variável> receberá <valor 1> se <condição> for verdadeira e <valor 2> caso contrário.

Operadores lógicos

Os operadores lógicos são: *and*, *or*, *not*, *is* e *in*.

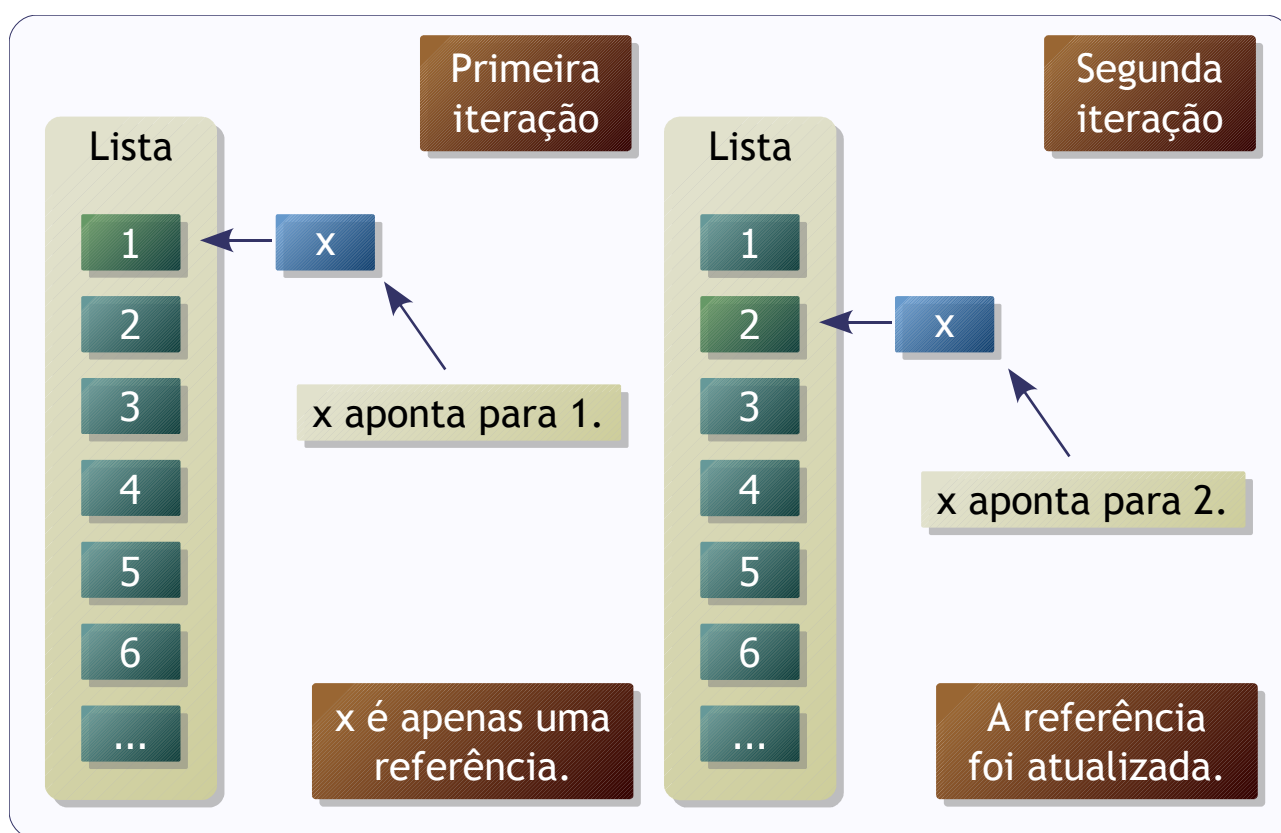
- *and*: retorna verdadeiro se e somente se receber duas expressões que forem verdadeiras.
- *or*: retorna falso se e somente se receber duas expressões que forem falsas.
- *not*: retorna falso se receber uma expressão verdadeira e vice-versa.
- *is*: retorna verdadeiro se receber duas referências ao mesmo objeto e falso em caso contrário.
- *in*: retorna verdadeiro se receber um item e uma lista e o item ocorrer uma ou mais vezes na lista e falso em caso contrário.

Com operadores lógicos é possível construir condições mais complexas para controlar desvios condicionais e laços.

Laços

Laços são estruturas de repetição, geralmente usados para processar coleções de dados, tais como linhas de um arquivo ou registros de um banco de dados, que precisam ser processados por um mesmo bloco de código.

For

É a estrutura de repetição mais usada no Python. Útil para percorrer seqüências ou processar iteradores.

A instrução *for* aceita não só seqüências estáticas, mas também seqüências geradas por iteradores. Iterador é uma estrutura que permite iterações, ou seja, acesso aos itens de uma coleção de elementos, de forma seqüencial.

Durante a execução de um laço *for*, a referência aponta para um elemento da seqüência. A cada iteração, a referência é atualizada, para que o bloco de código do *for* processe o elemento correspondente.

A clausula *break* interrompe o laço e *continue* passa para a próxima iteração. O código dentro do *else* é executado ao final do laço, a não ser que o laço tenha sido interrompido

por *break*.

Muito útil em laços com a instrução *for*, a função *range(m, n, p)* retorna uma lista de inteiros, começando em *m* e menores que *n*, em passos de comprimento *p*, que podem ser usados como seqüência para o laço.

Sintaxe:

```
for <referência> in <sequência>:
    <bloco de código>
    continue
    break
else:
    <bloco de código>
```

Exemplo:

```
# Soma de 0 a 99
s = 0
for x in range(1, 100):
    s = s + x
print s
```

O resultado é 4950.

While

Executa um bloco de código atendendo a uma condição.

Sintaxe:

```
while <condição>:
    <bloco de código>
    continue
    break
else:
    <bloco de código>
```

O bloco de código dentro do laço *while* é repetido enquanto a condição do laço estiver sendo avaliada como verdadeira.

Exemplo:

```
# Soma de 0 a 99
s = 0
x = 1

while x < 100:
   s = s + x
   x = x + 1
print s
```

O laço *while* é adequado quando não há como determinar quantas iterações vão ocorrer e não há uma seqüência a seguir.

Tipos

Variáveis no interpretador Python são criadas através da atribuição e destruídas pelo coletor de lixo, quando não existem mais referências a elas.

Os nomes das variáveis devem começar com letra (sem acentuação) ou sublinhado (_) e seguido por letras (sem acentuação), dígitos ou sublinhados (_), sendo que maiúsculas e minúsculas são consideradas diferentes.

Existem vários tipos simples de dados pré-definidos no Python, tais como:

- Números (inteiros, reais, complexos, ...).
- Texto.
- Booleanos.

Além disso, existem tipos que funcionam como coleções. Os principais são:

- Lista.
- Tupla.
- Dicionário.

Os tipos no Python podem ser:

- Mutáveis: permitem que os conteúdos das variáveis sejam alterados.
- Imutáveis: não permitem que os conteúdos das variáveis sejam alterados.

Em Python, os nomes de variáveis são referências, que podem ser alteradas em tempos de execução.

Números

Python oferece vários tipos numéricos:

- Inteiro (*int*): i = 1
- Real de ponto flutuante (*float*): f = 3.14
- Complexo (*complex*): c = 3 + 4j

Observações:

- As conversões entre inteiro e longo são automáticas.
- Reais podem ser representados em notação cientifica: 1.2e22

Operações numéricas:

- Soma (+).
- Diferença (-).
- Multiplicação (*).
- Divisão (/): entre dois inteiros funciona igual à divisão inteira. Em outros casos, o resultado é real.
- Divisão inteira (//): o resultado é truncado para o inteiro imediatamente inferior, mesmo quando aplicado em números reais, porém neste caso o resultado será real também.
- Exponenciação (**): pode ser usada para calcular a raiz, através de expoentes fracionários (exemplo: 100**0.5).
- Módulo (%): retorna o resto da divisão.

Durante as operações, os números serão convertidos de forma adequada (exemplo: (1.5+4j) + 3 resulta em 4.5+4j).

Além dos tipos *builtins* do interpretador, na biblioteca padrão ainda existe o módulo *decimal*, que define operações com números reais com precisão fixa, e tipos para data e hora.

Texto

As *strings* no Python são *buitins* para armazenar texto. São imutáveis, sendo assim, não é possível adicionar, remover ou mesmo modificar algum caractere de uma *string*. Para realizar essas operações, o Python precisa criar um nova *string*.

Tipos:

- *String* padrão: s = 'Led Zeppelin'
- *String unicode*: u = u'Björk'

A *string* padrão pode ser convertida para *unicode* através da função *unicode()*.

A inicialização de *strings* pode ser:

- Com aspas simples ou duplas.
- Em várias linhas consecutivas, desde que seja entre três aspas simples ou duplas.
- Sem expansão de caracteres (exemplo: s = r'\n', aonde s conterá os caracteres “\” e “n”).

Operações com *strings*:

```
s = 'Camel'

# Concatenação
print 'The ' + s + ' run away!'

# Interpolação
print 'tamanho de %s => %d' % (s, len(s))

# String tratada como seqüência
for ch in s: print ch

# Strings são objetos
if s.startswith('C'): print s.upper()

# o que acontecerá?
print 3 * s
# 3 * s é consistente com s +s + s
```

Interpolação:

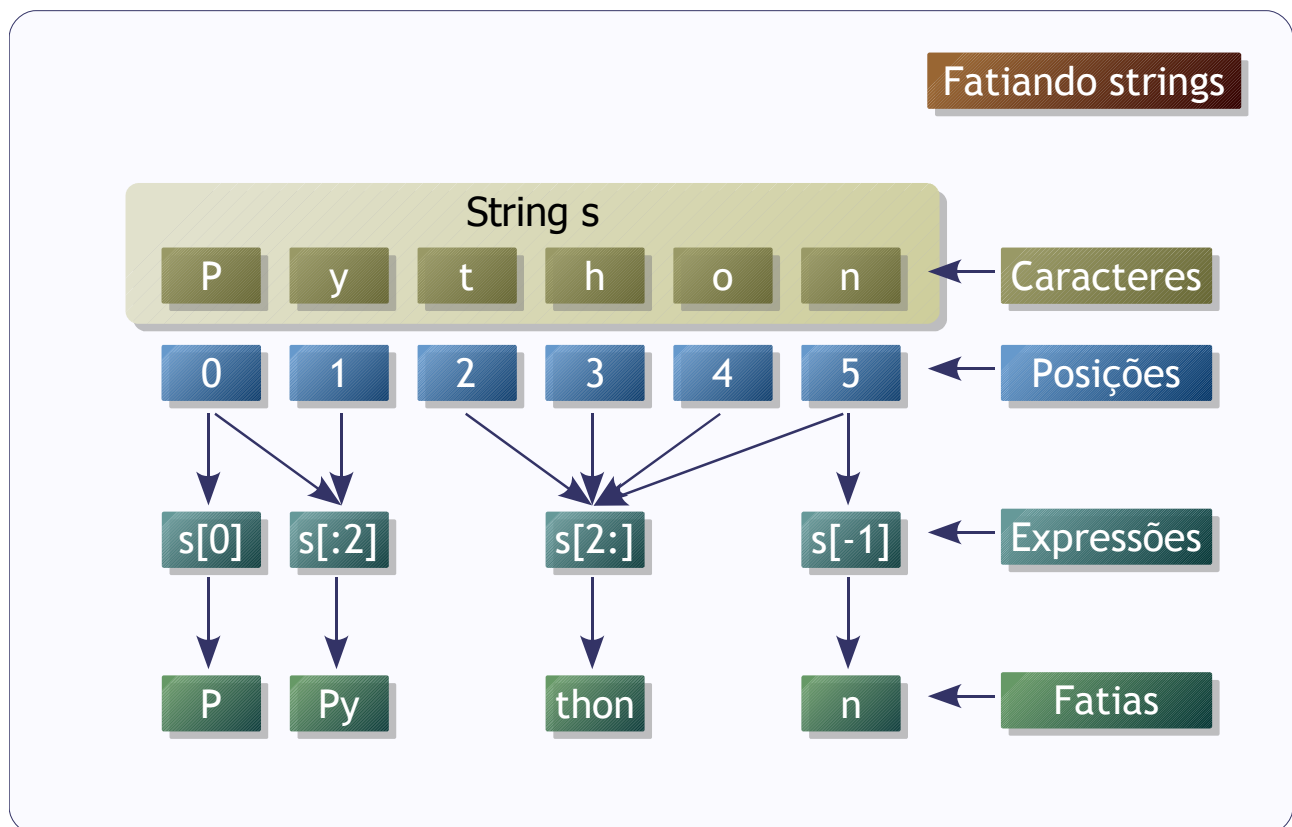
Operador “%” é usado para fazer interpolação de *strings*. A interpolação é mais eficiente do que a concatenação convencional.

```
print 'Agora são %02d:%02d.' % (16, 30)
```

```
# Imprime: Agora são 16:30.
```

Símbolos usados na interpolação:

- %s: *string*.
- %d: inteiro.
- %f: real.

Fatias (*slices*) de *strings* podem ser obtidas colocando índices entre colchetes após a *string*.

Os índices no Python:

- Começam em zero.
- Contam a partir do fim se forem negativos.
- Podem ser definidos como trechos, na forma [inicio:fim + 1:intervalo]. Se não for definido o inicio, será considerado como zero. Se não for definido o fim + 1, será considerado o tamanho do objeto. O intervalo (entre os caracteres), se não for definido, será 1.

É possível inverter *strings* usando um intervalo negativo:

```
print 'Python'[::-1]
# Mostra: nohtyP
```

Várias funções para tratar com texto estão no módulo *string*.

```
# -*- coding: latin1 -*-

# importando o módulo string
import string

# O alfabeto
a = string.ascii_letters

# Rodando o alfabeto um caractere para a esquerda
b = a[1:] + a[0]

# A função maketrans() cria uma tabela de tradução
# entre os caracteres das duas strings que ela
# recebeu como parâmetro.
# Os caracteres ausentes nas tabelas serão
# copiados para a saída.
tab = string.maketrans(a, b)

# A mensagem...
msg = '''Esse texto será traduzido..
Vai ficar bem estranho.
'''
# A função translate() usa a tabela de tradução
# criada pela maketrans() para traduzir uma string
print string.translate(msg, tab)
```

Saída:

```
Fttf ufyup tfsá usbevAjep..
Wbj gjdbs cfn ftusboip.
```

O módulo também implementa um tipo chamado *Template*, que é um modelo de *string* que pode ser preenchido através de um dicionário. Os identificadores são inciados por cifrão ($) e podem ser cercados por chaves, para evitar confusões.

Exemplo:

```
# -*- coding: latin1 -*-
```

```
# importando o módulo string
import string

# Cria uma string template
st = string.Template('$aviso aconteceu em $quando')

# Preenche o modelo com um dicionário
s = st.substitute({'aviso': 'Falta de eletricidade',
   'quando': '03 de Abril de 2002'})

# Mostra:
# Falta de eletricidade aconteceu em 03 de Abril de 2002
print s
```

É possível usar strings mutáveis no Python, através do módulo *UserString*, que define o tipo *MutableString*:

```
# -*- coding: latin1 -*-
# importando o módulo UserString
import UserString

s = UserString.MutableString('Python')
s[0] = 'p'

print s # mostra "python"
```

Strings mutáveis são menos eficientes do que *strings* mutáveis.

Listas

Listas são coleções heterogêneas de objetos, que podem ser de qualquer tipo, inclusive outras listas.

As listas no Python são mutáveis, podendo ser alteradas a qualquer momento. Listas podem ser fatiadas da mesma forma que as *strings*, mas como as listas são mutáveis, é possível fazer atribuições a itens da lista.

Sintaxe:

```
lista = [a, b, ..., z]
```

Operações comuns com listas:

```
# Uma nova lista: Brit Progs dos anos 70
progs = ['Yes', 'Genesis', 'Pink Floyd', 'ELP']

# Varrendo a lista inteira
for prog in progs:
    print prog

# Trocando o último elemento
progs[-1] = 'King Crimson'

# Incluindo
progs.append('Camel')

# Removendo
progs.remove('Pink Floyd')

# Ordena a lista
progs.sort()

# Inverte a lista
progs.reverse()

# Imprime numerado
for i, prog in enumerate(progs):
    print i + 1, '=>', prog

# Imprime do segundo item em diante
print progs[1:]

print range(10)
```

A função *enumerate()* retorna uma tupla de dois elementos a cada iteração: um número seqüencial e um item da seqüência correspondente.

A operações de ordenação (*sort*) e inversão (*reverse*) são realizadas na própria lista, sendo assim, não geram novas listas.

Tuplas

Semelhantes as listas, porém são imutáveis: não se pode acrescentar, apagar ou fazer atribuições aos itens.

Sintaxe:

```
tupla = (a, b, ..., z)
```

Os parênteses são opcionais.

Particularidade: tupla com apenas um elemento é representada como:

```
t1 = (1,)
```

Os elementos de uma tupla podem ser referenciados da mesma forma que os elementos de uma lista:

```
primeiro_elemento = tupla[0]
```

Listas podem ser convertidas em tuplas:

```
tupla = tuple(lista)
```

E tuplas podem ser convertidas em listas:

```
lista = list(tupla)
```

Embora a tupla possa conter elementos mutáveis, esses elementos não podem sofrer atribuição, pois isto modificaria a referência ao objeto.

Exemplo usando o modo interativo:

```
>>> t = ([1, 2], 4)
>>> t[0].append(3)
>>> t
([1, 2, 3], 4)
>>> t[0] = [1, 2, 3]
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

As tuplas são mais eficientes do que as listas convencionais, pois consomem menos recursos computacionais.

Outros tipos de seqüências

O Python provê entre os *builtins* também:

- *set*: lista mutável unívoca (sem repetições).
- *frozenset*: lista imutável unívoca.

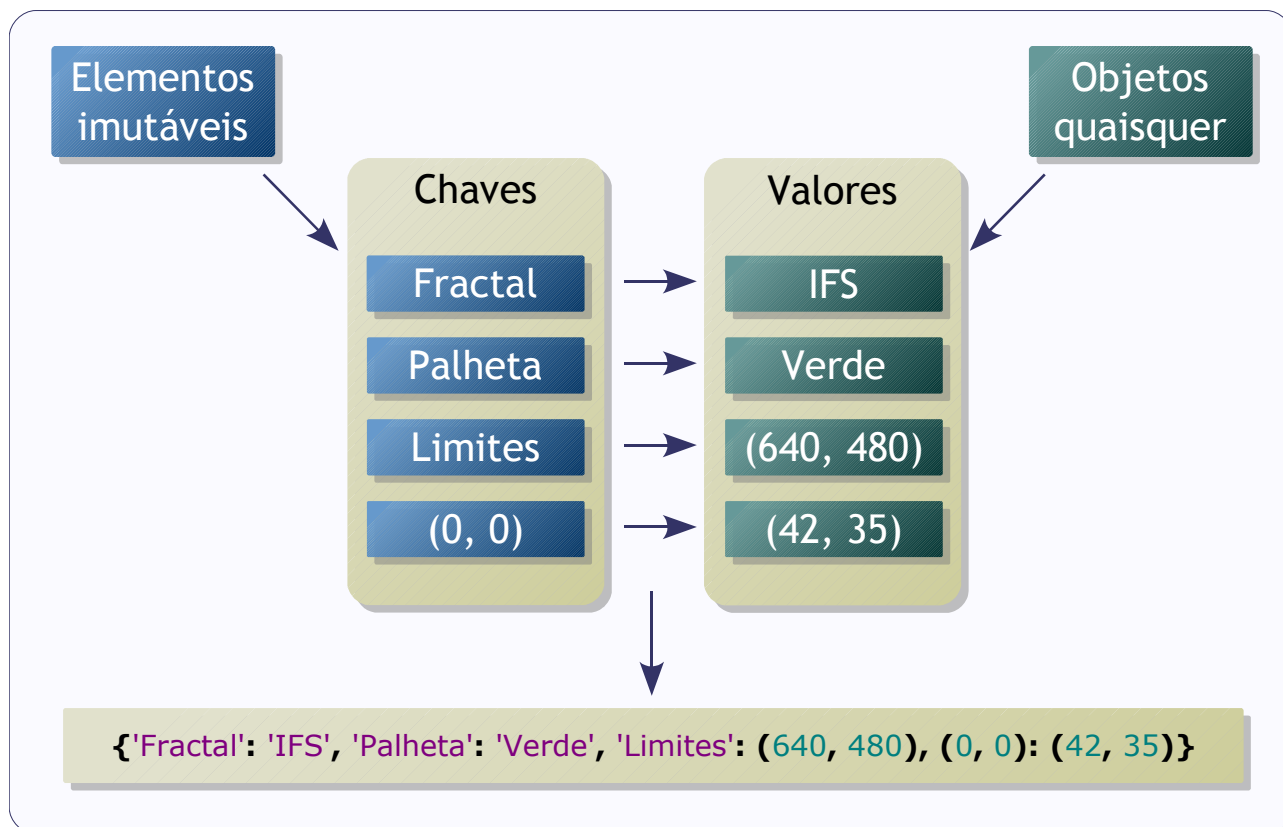
Dicionários

Um dicionário é uma lista de associações compostas por uma chave única e estruturas correspondentes. Dicionários são mutáveis, tais como as listas.

A chave tem que ser de um tipo imutável, geralmente *strings*, mas também podem ser tuplas ou tipos numéricos. O dicionário do Python não fornece garantia de que as chaves estarão ordenadas.

Sintaxe:

```
dicionario = {'a': a, 'b': b, ..., 'z', z}
```

Estrutura:

Exemplo de dicionário:

```
dic = {'nome': 'Shirley Manson', 'banda': 'Garbage'}
```

Acessando elementos:

```
print dic['nome']
```

Adicionando elementos:

```
dic['album'] = 'Version 2.0'
```

Apagando um elemento do dicionário:

```
del dic['album']
```

Obtendo os itens, chaves e valores:

```
itens = dic.items()
chaves = dic.keys()
valores = dic.values()
```

Exemplos com dicionários:

```
# Progs e seus albuns
progs = {'Yes': ['Close To The Edge', 'Fragile'],
    'Genesis': ['Foxtrot', 'The Nursery Crime'],
    'ELP': ['Brain Salad Surgery']}

# Mais progs
progs['King Crimson'] = ['Red', 'Discipline']

# Iteração
for prog, albuns in progs.items():
    print prog, '=>', albuns

# Se tiver 'ELP', deleta
if progs.has_key('ELP'):
    del progs['ELP']
```

items() retorna uma lista de tuplas com a chave e o valor.

Exemplo de matriz esparsa:

```
# -*- coding: latin1 -*-

# Matriz esparsa implementada
```

```
# com dicionário

# Matriz esparsa é uma estrutura
# que só armazena os valores que
# existem na matriz

dim = 6, 12
mat = {}

# Tuplas são imutáveis
# Cada tupla representa
# uma posição na matriz
mat[3, 7] = 3
mat[4, 6] = 5
mat[6, 3] = 7
mat[5, 4] = 6
mat[2, 9] = 4
mat[1, 0] = 9

for lin in range(dim[0]):
    for col in range(dim[1]):
        # Método get(chave, valor)
        # retorna o valor da chave
        # no dicionário ou se a chave
        # não existir, retorna o
        # segundo argumento
        print mat.get((lin, col), 0),
    print
```

Saída:

```
0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4 0 0
0 0 0 0 0 0 0 3 0 0 0 0
0 0 0 0 0 0 5 0 0 0 0 0
0 0 0 0 6 0 0 0 0 0 0 0
```

Gerando a matriz esparsa:

```
# -*- coding: latin1 -*-

# Matriz em forma de string
matriz = '''0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4 0 0
0 0 0 0 0 0 0 3 0 0 0 0
```

```
0 0 0 0 0 0 5 0 0 0 0 0
0 0 0 0 6 0 0 0 0 0 0 0'''

mat = {}

# Quebra a matriz em linhas
for lin, linha in enumerate(matriz.splitlines()):

   # Quebra a linha em colunas
   for col, coluna in enumerate(linha.split()):

      coluna = int(coluna)
      # Coloca a coluna no resultado,
      # se for diferente de zero
      if coluna:
         mat[lin, col] = coluna

print mat
# Some um nas dimensões pois a contagem começa em zero
print 'Tamanho da matriz completa:', (lin + 1) * (col + 1)
print 'Tamanho da matriz esparsa:', len(mat)
```

Saída:

```
{(5, 4): 6, (3, 7): 3, (1, 0): 9, (4, 6): 5, (2, 9): 4}
Tamanho da matriz completa: 72
Tamanho da matriz esparsa: 5
```

A matriz esparsa é uma boa solução de processamento para estruturas em que a maioria dos itens permanecem vazios, como planilhas, por exemplo.

Verdadeiro, falso e nulo

Em Python, o tipo booleano (*bool*) é uma especialização do tipo inteiro (*int*). O verdadeiro é chamado *True* e é igual a 1, enquanto o falso é chamado *False* e é igual a zero.

Os seguintes valores são considerados falsos:

- *False* (falso).
- *None* (nulo).
- 0 (zero).
- '' (*string* vazia).
- [] (lista vazia).
- () (tupla vazia).
- {} (dicionário vazio).
- Outras estruturas com o tamanho igual a zero.

São considerados verdadeiros todos os outros objetos fora dessa lista.

O objeto *None*, que é do tipo *NoneType*, do Python representa o nulo e é avaliado como falso pelo interpretador.

Funções

Funções são blocos de código identificados por um nome, que podem receber parâmetros pré-determinados.

No Python, as funções:

- Podem retornar ou não objetos.
- Aceitam *Doc Strings*.
- Aceitam parâmetros opcionais (com *defaults*). Se não for passado o parâmetro será igual ao *default* definido na função.
- Aceitam que os parâmetros sejam passados com nome. Neste caso,a ordem em que os parâmetros foram passados não importa.
- Tem *namespace* próprio (escopo local), e por isso podem ofuscar definições de escopo global.
- Podem ter suas propriedades alteradas (geralmente por decoradores).

Doc Strings são *strings* que estão associadas a uma estrutura do Python. Nas funções, as *Doc Strings* são colocadas dentro do corpo da função, geralmente no começo. O objetivo das *Doc Strings* é servir de documentação para aquela estrutura.

Sintaxe:

```
def func(parametro1, parametro2=padrao):
    """Doc String
    """
    <bloco de código>
    return valor
```

O retorno é opcional.

Os parâmetros com *default* devem ficar após os que não tem *default*.

Exemplo (fatorial):

```
def fatorial(num):

    if num <= 1:
        return 1
    else:
        return(num * fatorial(num - 1))
```

A função é recursiva.

Exemplo (série de Fibonacci):

```
def fib(n):
    """Fibonacci:
    fib(n) => fib(n - 1) + fib(n - 2) se n > 1
    fib(n) => 1 se n <= 1
    """
    if n > 1:
        return fib(n - 1) + fib(n - 2)
    else:
        return 1

# Mostrar Fibonacci de 1 a 5
for i in [1, 2, 3, 4, 5]:
    print i, '=>', fib(i)
```

Exemplo (conversão de RGB):

```
# -*- coding: latin1 -*-

def rgb_html(r=0, g=0, b=0):
    """Converte R, G, B em #RRGGBB"""

    return '#%02x%02x%02x' % (r, g, b)

def html_rgb(color='#000000'):
    """Converte #RRGGBB em R, G, B"""

    if color.startswith('#'): color = color[1:]

    r = int(color[:2], 16)
    g = int(color[2:4], 16)
    b = int(color[4:], 16)

    return r, g, b # Uma seqüência

print rgb_html(200, 200, 255)
print rgb_html(b=200, g=200, r=255) # O que houve?
print html_rgb('#c8c8ff')
```

Observações:

- Os argumentos com padrão devem vir por último, depois dos argumentos sem padrão.
- O valor do padrão para um parâmetro é calculado quando a função é definida.
- Os argumentos passados sem identificador são recebidos pela função na forma de uma lista.
- Os argumentos passados com identificador são recebidos pela função na forma de um dicionário.
- Os parâmetros passados com identificador na chamada da função devem vir no fim

da lista de parâmetros.

Exemplo de como receber todos parâmetros:

```
# -*- coding: latin1 -*-
# *args - argumentos sem nome (lista)
# **kargs - argumentos com nome (dicionário)

def func(*args, **kargs):
   print args
   print kargs
```

No exemplo, *kargs* receberá os argumentos nomeados e *args* receberá os outros.

Documentação

PyDOC é a ferramenta padrão de documentação[9] do Python. Ela pode ser utilizada tanto para acessar a documentação dos módulos que acompanham o Python, quanto módulos de terceiros.

No Windows, acesse o ícone "Module Docs" para a documentação da biblioteca padrão e "Python Manuals" para consultar o tutorial, referências e outros documentos mais extensos.

Para utilizar o PyDOC no Linux:

```
pydoc ./modulo.py
```

Para exibir a documentação de "modulo.py" no diretório atual.

No Linux, para ver a documentação das bibliotecas no *browser*, utilize o comando:

```
pydoc -p 8000
```

E acesse o endereço http://localhost:8000/.

Para rodar a versão gráfica do PyDOC execute:

```
pydoc -g
```

O PyDOC utiliza as *Doc Strings* dos módulos para gerar a documentação.

Além disso, é possível ainda consultar a documentação no próprio interpretador, através da função *help()*.

Exemplo:

```
help(list)
```

Mostra a documentação para a lista do Python.

9 A documentação do Python também disponível na internet em: http://www.python.org/doc/.

Exercícios I

1. Implementar duas funções:
   - Uma que converta temperatura em graus *Celsius* para *Fahrenheit.*
   - Outra que converta temperatura em graus *Fahrenheit* para *Celsius*.

Lembrando que:

$$F = \frac{9}{5} \cdot C + 32$$

2. Implementar uma função que retorne verdadeiro se o número for primo (falso caso contrário). Testar de 1 a 100.

3. Implementar uma função que receba uma lista de listas de comprimentos quaisquer e retorne uma lista de uma dimensão.

4. Implementar uma função que receba um dicionário e retorne a soma, a média e a variação dos valores.

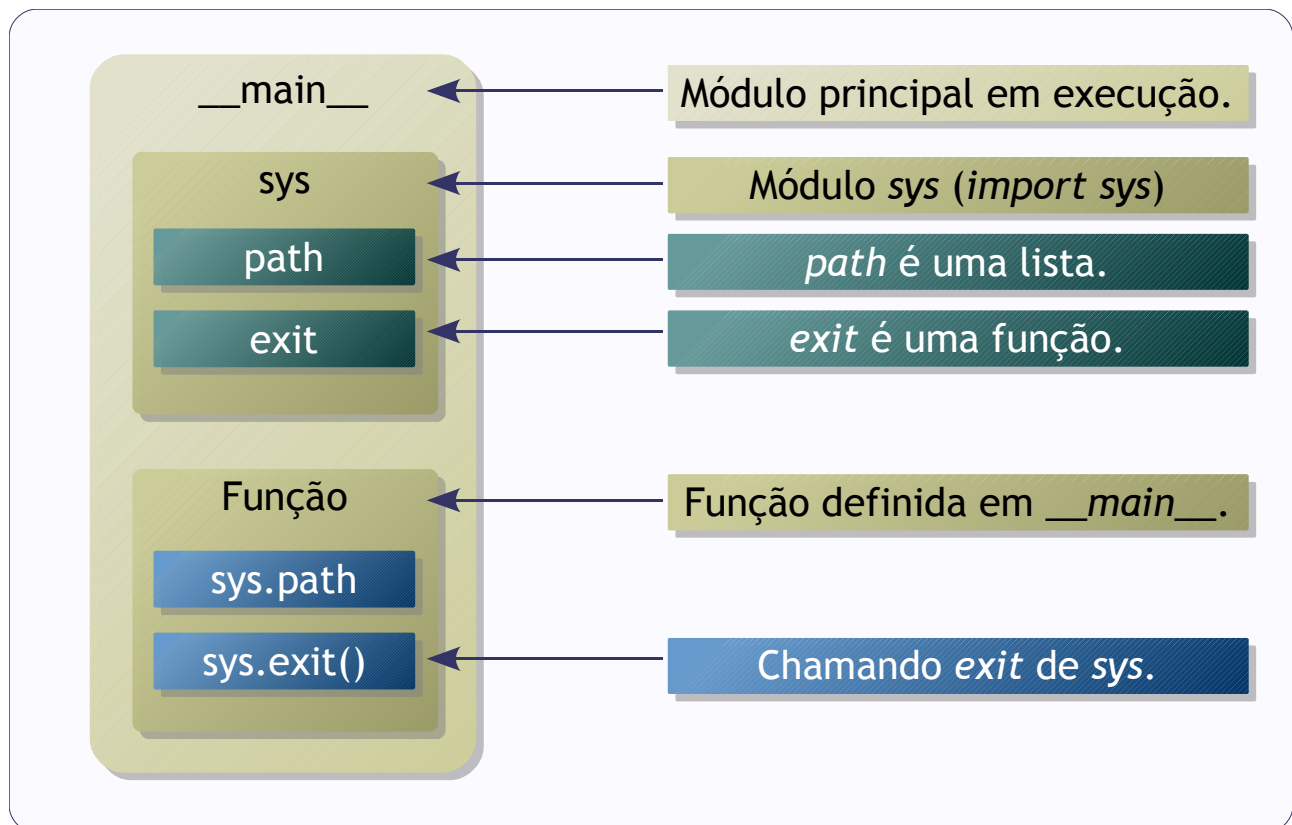
5. Escreva uma função que:
   - Receba uma frase como parâmetro.
   - Retorne uma nova frase com cada palavra com as letras invertidas.

6. Crie uma função que:
   - Receba uma lista de tuplas (dados), um inteiro (chave, zero por padrão igual) e um booleano (reverso, falso por padrão).
   - Retorne dados ordenados pelo item indicado pela chave e em ordem decrescente se reverso for verdadeiro.

Parte II

- Módulos.
- Escopo de nomes.
- Pacotes.
- Bibliotecas compartilhadas.
- Bibliotecas de terceiros.
- Arquivos e I/O.
- Exceções.
- Tempo.
- Introspecção.
- Exercícios II.

Módulos

Módulos são arquivos que podem importados para um programa. Podem conter qualquer estrutura do Python e são executados quando importados[10]. Eles são compilados quando importados pela primeira vez e armazenados em arquivo (com extensão *".pyc"* ou *".pyo"*), possuem *namespace* próprio e aceitam *Doc Strings*. São objetos *Singleton* (só é carregada uma instância em memória, que fica disponível de forma global para o programa).

Os módulos são localizados pelo interpretador através da lista de pastas *PYTHONPATH* (*sys.path*), que normalmente inclui a pasta corrente em primeiro lugar. O módulo principal de um programa tem a variável *__name__* igual à "__main__", então é possível testar se o módulo é o principal usando:

```
if __name__ == "__main__":
    # Aqui o código só será executado
    # se este for o módulo principal
    # e não quando ele for importado por outro programa
```

10 Caso seja necessário executar de novo o módulo durante a execução da aplicação, ele terá que carregado outra vez através da função *reload*.

Com isso é fácil transformar um programa em um módulo.

Os módulos são carregados através da instrução *import*. Desta forma, ao usar alguma estrutura do módulo, é necessário identificar o módulo. Isto é chamado de *importação absoluta*.

```
import os
print os.name
```

Também possível importar módulos de forma relativa:

```
from os import name
print name
```

O caractere "*" pode ser usado para importar tudo que está definido no módulo:

```
from os import *
print name
```

Por evitar problemas como a ofuscação de variáveis, a importação absoluta é uma prática melhor de programação do que a importação relativa.

Exemplo de módulo:

```
# -*- coding: latin1 -*-
"""
modutils => rotinas utilitárias para módulos
"""

import os.path
import sys
import glob

def find(txt):
    """encontra módulos que tem o nome
    contendo o parâmetro
    """

    resp = []

    for path in sys.path:
        mods = glob.glob('%s/*.py' % path)
```

```
        for mod in mods:
            if txt in os.path.basename(mod):
                resp.append(mod)

    return resp
```

Exemplo de uso do módulo:

```
from os.path import getsize, getmtime
from time import localtime, asctime

import modutils

mods = modutils.find('xml')

for mod in mods:

    tm = asctime(localtime(getmtime(mod)))
    kb = getsize(mod) / 1024
    print '%s: (%d kbytes, %s)' % (mod, kb, tm)
```

Saída:

```
c:\python25\lib\xmllib.py: (34 kbytes, Fri Oct 28 20:07:40 2005)
c:\python25\lib\xmlrpclib.py: (46 kbytes, Thu Dec 08 14:20:04 2005)
```

Dividir programas em módulos facilita o reaproveitamento e localização de falhas no código.

Módulos da biblioteca padrão

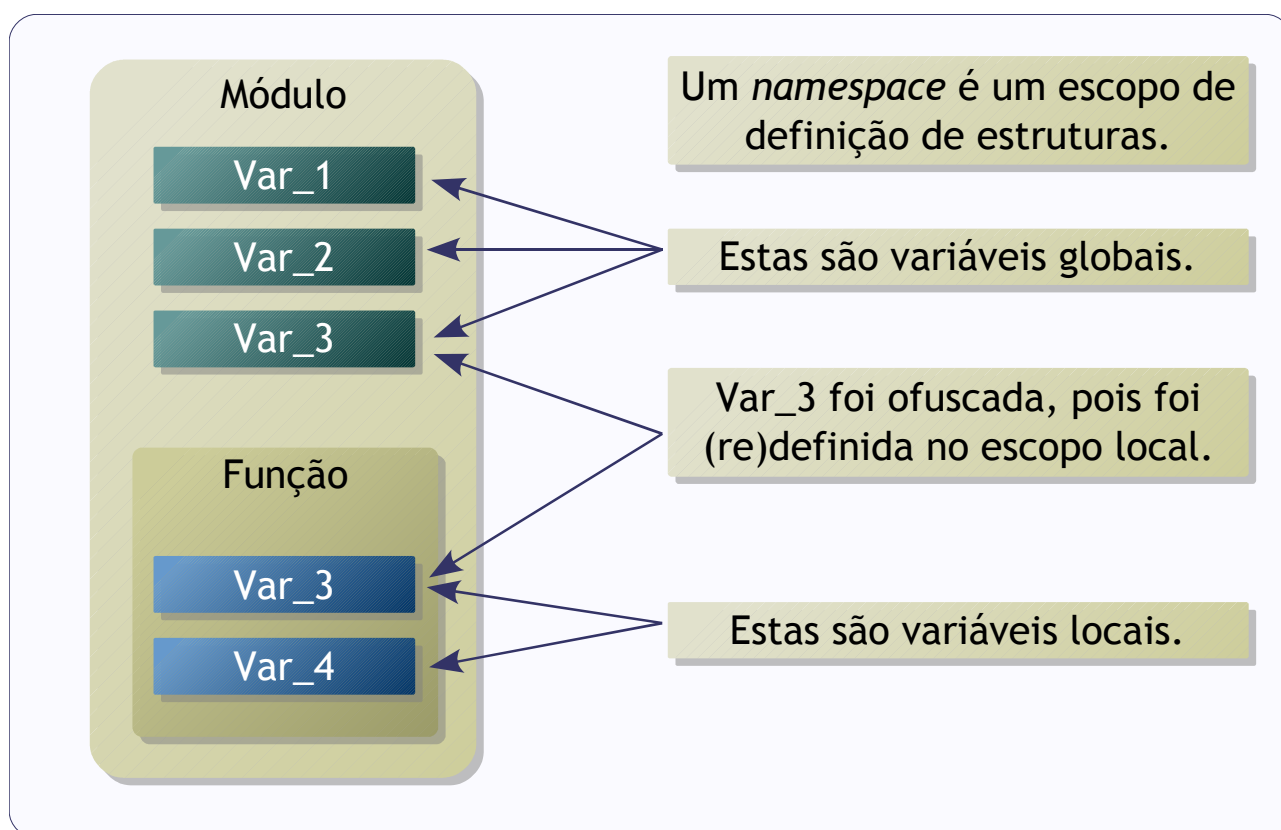
É comum dizer que o Python vem com "baterias inclusas", em referência a vasta biblioteca de módulos e pacotes que é distribuída com o interpretador.

Alguns módulos importantes da biblioteca padrão:

- Sistema: *os, glob, shutils, subprocess*.
- *Threads*: *threading*.
- Persistência: *pickle, cPickle*.
- XML: *xml.dom, xml.sax, elementTree* (a partir da versão 2.5).
- Configuração: *ConfigParser, optparse*.
- Tempo: *time, datetime*.
- Outros: *sys, logging, traceback, types, timeit*.

Escopo de nomes

Namespaces são dicionários que relacionam os nomes dos objetos (referências) e os objetos em si.

Normalmente, os nomes estão definidos em dois dicionários, que podem ser consultados através das funções *locals()* e *globals()*. Estes dicionários são alterados dinamicamente em tempo de execução[11].

Variáveis globais podem ser ofuscadas por variáveis (pois o escopo local é verificado antes do escopo global). Para evitar isso, é preciso declarar a variável como global no escopo local.

Exemplo:

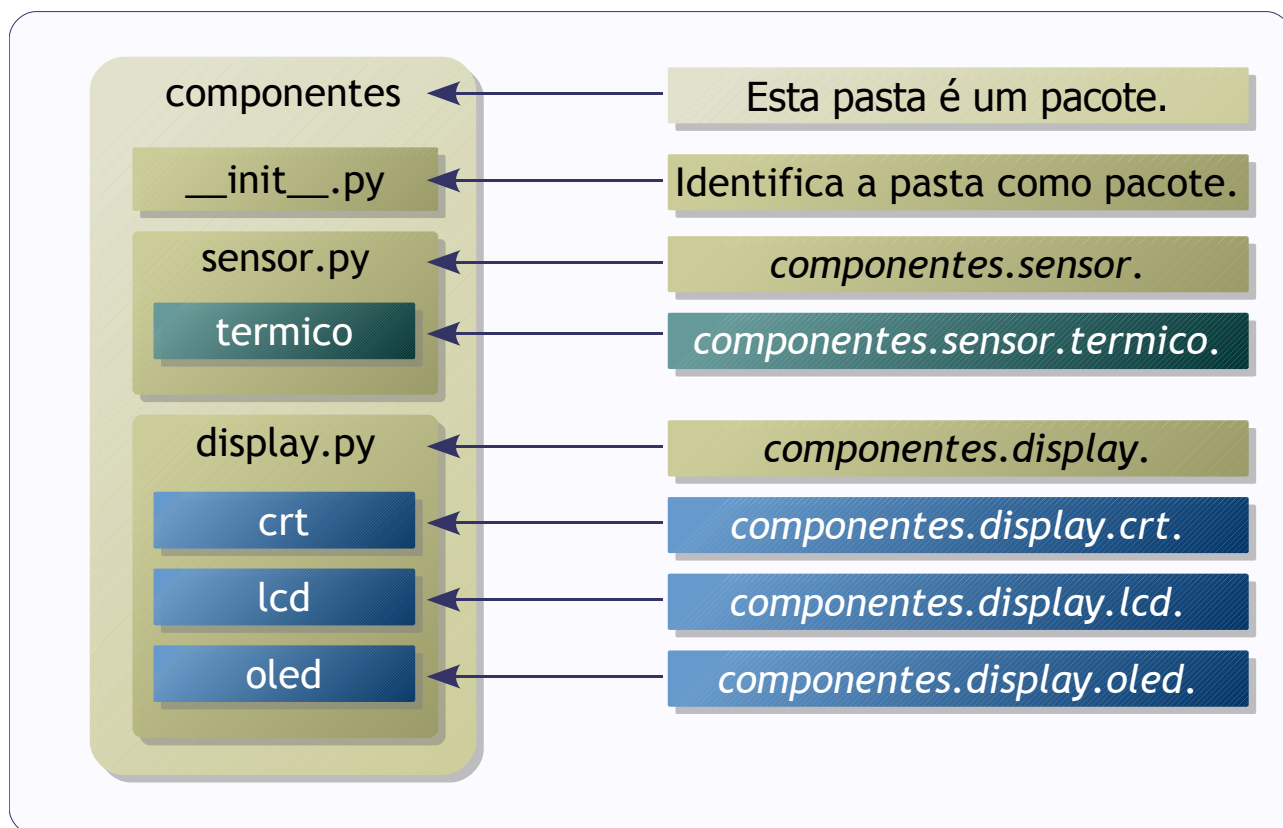
```
def somalista(lista):
    """
```

11 Embora os dicionários retornados por *locals()* e *globals()* possam ser alterados diretamente, isso deve ser evitado, pois pode ter efeitos indesejáveis.

```
    Soma listas de listas, recursivamente
    Coloca o resultado como global
    """
    global soma

    for item in lista:
        if type(item) is list: # Se o tipo do item for lista
            somalista(item)
        else:
            soma += item

soma = 0
somalista([[1, 2], [3, 4, 5], 6])

print soma # 21
```

Usar variáveis globais não é considerada uma boa prática de desenvolvimento, pois tornam mais difícil entender o sistema, portanto é melhor evitar seu uso. E ofuscar variáveis também.

Pacotes

Pacotes (*packages*) são pastas que são identificadas pelo interpretador pela presença de um arquivo com o nome "__init__.py". Os pacotes servem como *containers* para módulos.

É possível importar todos os módulos do pacote usando a declaração *from nome_do_pacote import* *.

O arquivo "__init__.py" pode estar vazio ou conter código de inicialização do pacote ou definir uma variável chamada *__all__*, uma lista de módulos do pacote serão importados quando for usado "*". Sem o arquivo, o Python não identifica a pasta como um pacote válido.

Bibliotecas compartilhadas

A partir da versão 2.5, o Python incorporou o módulo *ctypes*, que implementa tipos compatíveis com os tipos usados pela linguagem C e permite evocar funções de bibliotecas compartilhadas.

O módulo provê várias formas de evocar funções. Funções que seguem a convenção de chamada *stdcall*, como a API do Windows, podem ser acessadas através da classe *windll*. *Dynamic-link library* (DLL) é a implementação de bibliotecas compartilhadas que são usadas no Windows.

Exemplo com *windll*:

```
# -*- coding: latin1 -*-

import ctypes

# Evocando a caixa de mensagens do Windows
# Os argumentos são: janela pai, mensagem,
# título da janela e o tipo da janela.
# A função retorna um inteiro, que
# corresponde a que botão foi pressionado
i = ctypes.windll.user32.MessageBoxA(None,
    'Teste de DLL!', 'Mensagem', 0)

# O resultado indica qual botão foi clicado
print i
```

Para funções que seguem a convenção de chamada *cdecl*, usada pela maioria dos compiladores C, existe a classe *cdll*. Para as passagens de argumentos por referência é preciso criar uma variável que funciona como um *buffer* para receber os resultados. Isso é necessário para receber *strings*, por exemplo.

Exemplo com *cdll* e *buffer*:

```
# -*- coding: latin1 -*-

import ctypes

# msvcrt é a biblioteca com a maioria das funções
# padrões da linguagens C no Windows
# O Windows coloca automaticamente
# a extensão do arquivo
clib = ctypes.cdll.msvcrt
```

```
# Cria um buffer para receber o resultado
# a referência para o buffer será passada para
# a função, que preenche o buffer com o resultado
s = ctypes.c_buffer('\000', 40)

# sscanf() é uma função que extrai valores
# de uma string conforme uma mascara
clib.sscanf('Testando sscanf!\n',
    'Testando %s!\n', s)

# Mostra o resultado
print s.value
```

É possível também evocar funções de bibliotecas compartilhadas no Linux:

```
# -*- coding: latin1 -*-

import ctypes

# Carrega a biblioteca padrão C no Linux
# A extensão do arquivo precisa passada
# para a função LoadLibrary()
clib = ctypes.cdll.LoadLibrary("libc.so.6")

# Cria um buffer para receber o resultado
s = ctypes.c_buffer('\000', 40)

# Evoca a função sprintf
clib.sprintf(s, 'Testando %s\n', 'sprintf!')

# Mostra o resultado
print s.value
```

Através de bibliotecas compartilhadas é possível usar código desenvolvido em outras linguagens de uma maneira simples.

Bibliotecas de terceiros

Hoje existem muitas bibliotecas escritas por terceiros disponíveis para Python, compostas por pacotes ou módulos, que implementam diversos recursos além da biblioteca padrão.

Geralmente, as bibliotecas são distribuídas das seguintes formas:

- Pacotes *distutils*.
- Pacotes para gerenciadores de pacotes do Sistema Operacional.
- Instaladores.
- Python Eggs.

Os pacotes usando o módulo *distutils*, que é distribuído com o Python, são muito populares. Os pacotes são distribuídos em arquivos compactados (geralmente ".tar.gz", ".tar.bz2" ou ".zip"). Para instalar, é necessário descompactar o arquivo, entrar na pasta que foi descompactada e por fim executar o comando:

```
python setup.py install
```

Que o pacote será instalado na pasta "site-packages" no Python.

Gerenciadores de pacotes do Sistema Operacional, geralmente trabalham com formatos próprios de pacote, como ".deb" (Debian Linux) ou ".rpm" (RedHat Linux). A forma de instalar os pacotes depende do gerenciador utilizado. A grande vantagem é que o gerenciador de pacotes cuida das dependências e atualizações.

Instaladores são executáveis que instalam a biblioteca. Geralmente são usados em ambiente Windows e podem ser desinstalados pelo Painel de Controle.

Python Eggs são pacotes (com a extensão ".egg") administrados pelo easy_install, utilitário que faz parte do projeto setuptools[12]. Semelhante a algumas ferramentas encontradas em outras linguagens, como o Ruby Gems, aos poucos está se tornando o padrão de fato para distribuição de bibliotecas em Python.

O programa procura pela versão mais nova do pacote no PYPI[13] (*Python Package Index*), repositório de pacotes Python, e também procura instalar as dependências que forem necessárias.

12 Fontes e documentação do projeto em: http://peak.telecommunity.com/DevCenter/setuptools/.
13 Endereço: http://pypi.python.org/pypi.

Python Eggs podem ser instalados pelo comando:

```
easy_install nome_do_pacote
```

O *script* easy_install é instalado na pasta "scripts" do Python.

Arquivos e I/O

Os arquivos no Python são representados por objetos do tipo *file*[14], que oferecem métodos para diversas operações de arquivos. Arquivos podem ser abertos para leitura ('r', que é o *default*), gravação ('w') ou adição ('a'), em modo texto ou binário('b').

Em Python:

- *sys.stdin* representa a entrada padrão.
- *sys.stdout* representa a saída padrão.
- *sys.stderr* representa a saída de erro padrão.

A entrada, saída e erro padrões são tratados pelo Python como arquivos abertos. A entrada em modo de leitura e os outros em modo de gravação.

Exemplo de escrita:

```
import sys

# Criando um objeto do tipo file
temp = file('temp.txt', 'w')

# Escrevendo no arquivo
for i in range(100):
    temp.write('%03d\n' % i)

# Fechando
temp.close()

temp = file('temp.txt')

# Escrevendo no terminal
for x in temp:
    sys.stdout.write(x)
#Escrever em sys.stdout envia
# otexto para a saída padrão

temp.close()
```

A iteração do segundo laço, o objeto retorna uma linha do arquivo.

Exemplo de leitura:

```
import sys
```

14 A referência *open* aponta para *file*.

```
import os.path

# raw_input retorna a string digitada
fn = raw_input('Nome do arquivo: ').strip()

if not os.path.exists(fn):
    print 'Tente outra vez...'
    sys.exit()

# Numerando as linhas
for i, s in enumerate(file(fn)):
    print i + 1, s,
```

É possível ler todas as linhas com o método *readlines()*:

```
# Imprime uma lista contendo linhas do arquivo
print file('temp.txt').readlines()
```

Os objetos de arquivo também possuem um método *seek()*, que permite ir para qualquer posição no arquivo.

Sistema de arquivos

O módulo *os.path* implementa várias funcionalidades relacionadas a sistemas de arquivo, tais como:

- *os.path.basename()*: retorna o componente final de um caminho.
- *os.path.dirname()*: retorna um caminho sem o componente final.
- *os.path.exists()*: retorna *True* se o caminho existe ou *False* em caso contrário.
- *os.path.getsize()*: retorna o tamanho do arquivo em *bytes*.

O *glob* é outro módulo relacionado ao sistema de arquivo:

```
import os.path
import glob

# Mostra uma lista de nomes de arquivos
# e seus respectivos tamanhos
for arq in sorted(glob.glob('*.py')):
    print arq, os.path.getsize(arq)
```

A função *glob.glob()* retorna uma lista com os nomes de arquivo que atendem ao critério passado como parâmetro, de forma semelhante ao comando “ls” disponível nos sistemas UNIX.

Arquivos compactados

O Python possui módulos para trabalhar com vários formatos de arquivos compactados.

Exemplo de gravação de um arquivo ".zip":

```
# -*- coding: latin1 -*-
"""
Gravando texto em um arquivo compactado
"""

import zipfile

texto = """
***************************************
Esse é o texto que será compactado e...
... guardado dentro de um arquivo zip.
***************************************
"""

# Cria um zip novo
zip = zipfile.ZipFile('arq.zip', 'w',
    zipfile.ZIP_DEFLATED)

# Escreve uma string no zip como se fosse um arquivo
zip.writestr('texto.txt', texto)

# Fecha o zip
zip.close()
```

Exemplo de leitura:

```
# -*- coding: latin1 -*-
"""
Lendo um arquivo compactado
"""

import zipfile

# Abre o arquivo zip para leitura
zip = zipfile.ZipFile('arq.zip')

# Pega a lista dos arquivos compactados
arqs = zip.namelist()

for arq in arqs:
    # Mostra o nome do arquivo
    print 'Arquivo:', arq
```

```
# Informações do arquivo
zipinfo = zip.getinfo(arq)
print 'Tamanho original:', zipinfo.file_size
print 'Tamanho comprimido:', zipinfo.compress_size

# Mostra o conteúdo do arquivo
print zip.read(arq)
```

Saída:

```
Arquivo: texto.txt
Tamanho original: 160
Tamanho comprimido: 82

**************************************
Esse é o texto que será compactado e...
... guardado dentro de um arquivo zip.
**************************************
```

O Python também provê módulos para os formatos gzip, bzip2 e tar, que são bastante utilizados em ambientes UNIX.

Arquivos de dados

Na biblioteca padrão, o Python também fornece um módulo para simplificar o processamento de arquivos no formato CSV (*Comma Separated Values*).

O formato é muito simples, os dados são armazenados em forma de texto, separados por vírgula, um registro por linha.

Exemplo de escrita:

```
import csv

# Dados
dt = (('temperatura', 15.0, 'C', '10:40', '2006-12-31'),
    ('peso', 42.5, 'kg', '10:45', '2006-12-31'))

# A escrita recebe um objeto do tipo "file"
out = csv.writer(file('dt.csv', 'w'))

# Escrevendo as tuplas no arquivo
out.writerows(dt)
```

Arquivo de saída:

```
temperatura,15.0,C,10:40,2006-12-31
peso,42.5,kg,10:45,2006-12-31
```

Exemplo de leitura:

```
import csv

# A leitura recebe um objeto arquivo
dt = csv.reader(file('dt.csv'))

# Para cada registro do arquivo, imprima
for reg in dt:
   print reg
```

Saída:

```
['temperatura', '15.0', 'C', '10:40', '2006-12-31']
['peso', '42.5', 'kg', '10:45', '2006-12-31']
```

O formato CSV é aceito pela maioria das planilhas e sistemas de banco de dados para importação e exportação de informações.

Exceções

Quando ocorre uma falha no programa (como uma divisão por zero, por exemplo) em tempo de execução, uma exceção é gerada. Se a exceção não for tratada, ela será propagada através das chamadas de função até o módulo principal do programa, interrompendo a execução.

```
print 1/0

Traceback (most recent call last):
  File "<input>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

A instrução *try* permite o tratamento de exceções no Python. Se ocorrer uma exceção em um bloco marcado com *try*, é possível tratar a exceção através da instrução *except*. Podem existir vários *excepts* para um *try*.

```
try:
    print 1/0

except ZeroDivisionError:
    print 'Erro ao tentar dividir por zero.'

# Mostra:
# Erro ao tentar dividir por zero.
```

Se *except* recebe o nome da exceção, só esta será tratada. Se não for passada nenhuma exceção como parâmetro, todas serão tratadas.

```
import traceback

# Tente receber o nome do arquivo
try:
    fn = raw_input('Nome do arquivo: ').strip()

    # Numerando as linhas
    for i, s in enumerate(file(fn)):
        print i + 1, s,

# Se ocorrer um erro
except:

    # Mostre na tela
    trace = traceback.format_exc()
```

```
    # E salve num arquivo
    print 'Aconteceu um erro:\n', trace
    file('trace.log', 'a').write(trace)

    # Encerre o programa
    raise SystemExit
```

O módulo *traceback* oferece funções para manipular as mensagens de erro. A função *format_exc* retorna a saída da última exceção formatada em uma *string*.

O tratamento de exceções pode possuir um bloco *else*, que será executado quando não ocorrer nenhuma exceção e um bloco *finally*, será executado de qualquer forma, tendo ocorrido uma exceção ou não[15]. Novos tipos de exceções podem ser definidos através de herança a partir da classe *Exception*.

15 A declaração *finally* pode ser usada para liberar recursos que foram usados no bloco *try*, tais como conexões de banco de dados ou arquivos abertos.

Tempo

O Python possui dois módulos para lidar com tempo:

- *time*: implementa funções básicas para lidar com o tempo gerado pelo sistema operacional.
- *datetime*: implementa tipos de alto nível para lidar operações de data e hora.

Exemplo com time:

```
# -*- coding: latin-1 -*-

import time

# localtime() Retorna a data e hora local no formato
# de uma tupla:
# (ano, mês, dia, hora, minuto, segundo, dia da semana,
#   dia do ano, horário de verão)
print time.localtime()

# asctime() retorna a data e hora como string, conforme
# a configuração do sistema operacional
print time.asctime()

# time() retorna o tempo do sistema em segundos
ts1 = time.time()

# gmtime() converte segundos para tuplas no mesmo
# formato de localtime()
tt1 = time.gmtime(ts1)
print ts1, '=>', tt1

# Somando uma hora
tt2 = time.gmtime(ts1 + 3600.)

# mktime() converte tuplas para segundos
ts2 = time.mktime(tt2)
print ts2, '=>', tt2

# clock() retorma o tempo desde quando o programa
# iniciou, em segundos
print 'O programa levou', time.clock(), \
    'segundos sendo executado até agora...'

# Contando os segundos...
for i in xrange(5):

    # sleep() espera durante o número de segundos
    # especificados
    time.sleep(1)
```

```
    print i + 1, 'segundo(s)'
```

Saída:

```
(2008, 5, 11, 9, 55, 57, 6, 132, 0)
Sun May 11 09:55:57 2008
1210510557.44 => (2008, 5, 11, 12, 55, 57, 6, 132, 0)
1210524957.0 => (2008, 5, 11, 13, 55, 57, 6, 132, 0)
O programa levou 1.53650813162e-006 segundos sendo executado até agora...
1 segundo(s)
2 segundo(s)
3 segundo(s)
4 segundo(s)
5 segundo(s)
```

Em *datetime,* estão definidos quatro tipos para representar o tempo:

- *datetime*: data e hora.
- *date*: apenas data.
- *time*: apenas hora.
- *timedelta*: diferença entre tempos.

Exemplo:

```
# -*- coding: latin-1 -*-

import datetime

# datetime() recebe como parâmetros:
# ano, mês, dia, hora, minuto, segundo
# e retorna um objeto do tipo datetime
dt = datetime.datetime(2020, 12, 31, 23, 59, 59)

# Objetos date e time podem ser criados
# a partir de um objeto datetime
data = dt.date()
hora = dt.time()

# Quanto tempo falta para 31/12/2020
dd = dt - dt.today()

print 'Data:', data
print 'Hora:', hora
print 'Quanto tempo falta para 31/12/2020:', \
    str(dd).replace('days', 'dias')
```

Saída:

```
Data: 2020-12-31
Hora: 23:59:59
Quanto tempo falta para 31/12/2020: 4616 dias, 13:22:58.857000
```

Os objetos dos tipos *date* e *datetime* retornam datas em formato ISO.

Introspecção

Introspecção ou reflexão é capacidade do software de identificar e relatar suas próprias estruturas internas, tais como tipos, escopo de variáveis, métodos e atributos.

Funções nativas do interpretador para introspecção:

Função	Retorno
type(objeto)	O tipo (classe) do objeto.
id(objeto)	O identificador do objeto.
locals()	O dicionário de variáveis locais.
globals()	O dicionário de variáveis globais.
len(objeto)	O tamanho do objeto.
dir(objeto)	A lista de estruturas do objeto.
help(objeto)	As Doc Strings do objeto.
repr(objeto)	A representação do objeto.
isinstance(objeto, classe)	Verdadeiro se objeto deriva da classe.
issubclass(subclasse, classe)	Verdadeiro se subclasse herda classe.

O identificador do objeto é um número inteiro único que é usado pelo interpretador para identificar internamente os objetos.

Exemplo:

```
# -*- coding: latin1 -*-

# Colhendo algumas informações
# dos objetos globais no programa

from types import ModuleType

def info(n_obj):

    # Cria uma referência ao objeto
    obj = globals()[n_obj]

    # Mostra informações sobre o objeto
    print 'Nome do objeto:', n_obj
    print 'Identificador:', id(obj)
    print 'Tipo:', type(obj)
    print 'Representação:', repr(obj)
```

```
    # Se for um módulo
    if isinstance(obj, ModuleType):
        print 'itens:'
        for item in dir(obj):
            print item
    print

# Mostrando as informações
for n_obj in dir():
    info(n_obj)
```

O Python também tem um módulo chamado *types*, que tem as definições dos tipos básicos do interpretador.

Através da introspecção, é possível determinar os campos de uma tabela de banco de dados, por exemplo.

Módulo inspect

O módulo *inspect* provê um conjunto de funções de introspecção prontas que permitem investigar tipos , itens de coleções, classes, funções, código fonte e a pilha de execução do interpretador.

Exemplo:

```
# -*- coding: latin1 -*-

import os.path
# inspect: módulo de introspecção "amigável"
import inspect

print 'Objeto:', inspect.getmodule(os.path)

print 'Classe?', inspect.isclass(str)

# Lista todas as funções que existem em "os.path"

print 'Membros:',

for name, struct in inspect.getmembers(os.path):

    if inspect.isfunction(struct):
        print name,
```

Saída:

```
Objeto: <module 'ntpath' from 'C:\Python24\lib\ntpath.pyc'>
Classe? True
Membros: abspath basename commonprefix dirname exists expanduser expandvars getatime getctime getmtime getsize isabs isdir isfile islink ismount join lexists normcase normpath realpath split splitdrive splitext splitunc walk
```

As funções que trabalham com a pilha do interpretador devem ser usadas com cuidado, pois é possível criar referências cíclicas (uma variável que aponta para o item da pilha que tem a própria variável). A existência de referências a itens da pilha retarda a destruição dos itens pelo coletor de lixo do interpretador.

Exercícios II

1. Implementar um programa que receba um nome de arquivo e gere estatísticas sobre o arquivo (número de caracteres, número de linhas e número de palavras)

2. Implementar um módulo com duas funções:

- *matrix_sum(*matrices)*, que retorna a matriz soma de matrizes de duas dimensões.
- *camel_case(s)*, que converte nomes para CamelCase.

3. Implementar uma função que leia um arquivo e retorne uma lista de tuplas com os dados (o separador de campo do arquivo é vírgula), eliminando as linhas vazias. Caso ocorra algum problema, imprima uma mensagem de aviso e encerre o programa.

4. Implementar um módulo com duas funções:

- *split(fn, n)*, que quebra o arquivo fn em partes de n *bytes* e salva com nomes seqüenciais (se *fn* = arq.txt, então arq_001.txt, arq_002.txt, ...)
- *join(fn, fnlist)* que junte os arquivos da lista *fnlist* em um arquivo só *fn*.

5. Crie um *script* que:

- Compare a lista de arquivos em duas pastas distintas.
- Mostre os nomes dos arquivos que tem conteúdos diferentes e/ou que existem em apenas uma das pastas.

6. Faça um *script* que:

- Leia um arquivo texto.
- Conte as ocorrências de cada palavra.
- Mostre os resultados ordenados pelo número de ocorrências.

Parte III

- Geradores.
- Programação funcional.
- Exercícios III.

Geradores

As funções geralmente seguem o fluxo convencional de processar, retornar valores e encerrar. Geradores são estruturas semelhantes, porém processam e retornam um valor de uma seqüência a cada chamada. O gerador lembra o estado do processamento entre as chamadas, retornam o próximo item esperado.

Os geradores apresentam várias vantagens em relação às funções convencionais:

- *Lazy Evaluation*: geradores só são processados quando é realmente necessário, sendo assim, economizam recursos de processamento.
- Reduzem a necessidade da criação de listas.
- Permitem trabalhar com seqüências ilimitadas de elementos.

Geradores normalmente são evocados através de um laço *for*. A sintaxe é semelhante a da função tradicional, só que a instrução *yield* substitui o *return*. A nova cada iteração, *yield* retorna o próximo valor.

Exemplo:

```
# -*- coding: latin-1 -*-

def gen_pares():
    """
    Gera números pares infinitamente...
    """

    i = 0

    while True:

        i += 2
        yield i

# Mostra cada número e passa para o próximo
for n in gen_pares():
    print n
```

No gerador, *yield* toma o lugar do *return*.

Outro exemplo:

```
import os

# Encontra arquivos recursivamente
def find(path='.'):
```

```
for item in os.listdir(path):
    fn = os.path.normpath(os.path.join(path, item))

    if os.path.isdir(fn):

        for f in find(fn):
            yield f

    else:
        yield fn

for fn in find('c:/temp'):
    print fn
```

A cada iteração do laço, o gerador encontra um novo arquivo.

Na própria linguagem existem vários geradores, como o *builtin* *xrange()*[16], e o módulo *itertools*, que define vários geradores úteis.

Para converter a saída do gerador em uma lista:

```
lista = list(gerador())
```

Assim, todos os itens serão gerados de uma vez.

16 O gerador *xrange* substitui com vantagem a função *range* e a sintaxe é a mesma.

Programação funcional

Programação funcional é um paradigma que trata a computação como uma avaliação de funções matemáticas. Tais funções podem ser aplicadas em seqüências de dados (geralmente listas).

São exemplos de linguagens funcionais: LISP, Scheme e Haskell (esta última influenciou o projeto do Python de forma marcante).

As operações básicas do paradigma funcional são implementadas no Python pelas funções *map()*, *filter()*, *reduce()* e *zip()*.

Lambda

No Python, *lambda* é uma função anônima composta apenas por expressões. As funções *lambda* podem ter apenas uma linha, e podem ser atribuídas a uma variável. Funções *lambda* são muito usadas em programação funcional.

Sintaxe:

```
lambda <lista de variáveis>: <expressões >
```

Exemplo:

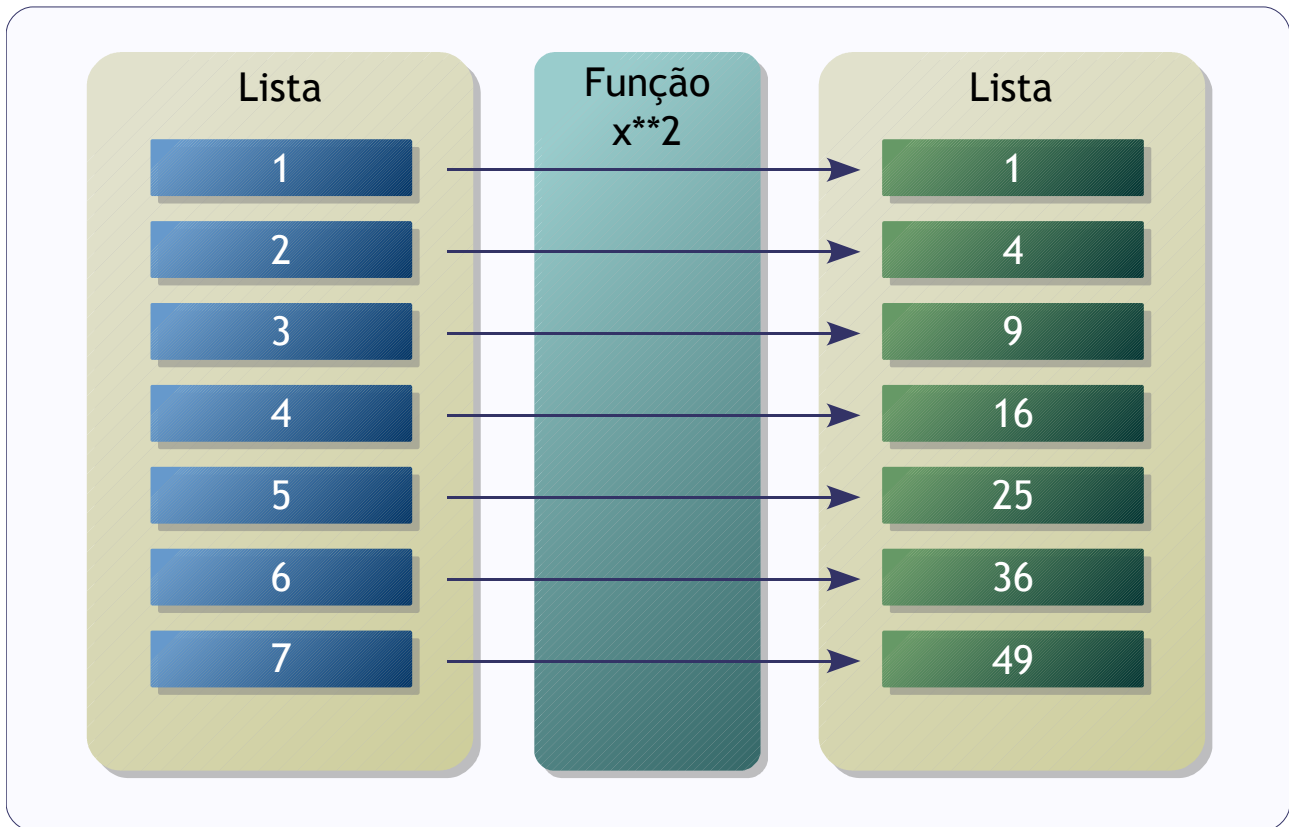
```
# Amplitude de um vetor 3D
amp = lambda x, y, z: (x ** 2 + y ** 2 + z ** 2) ** .5

print amp(1, 1, 1)
print amp(3, 4, 5)
```

Saída:

```
1.73205080757
7.07106781187
```

Funções *lambda* consomem menos recursos que as funções convencionais.

Mapeamento

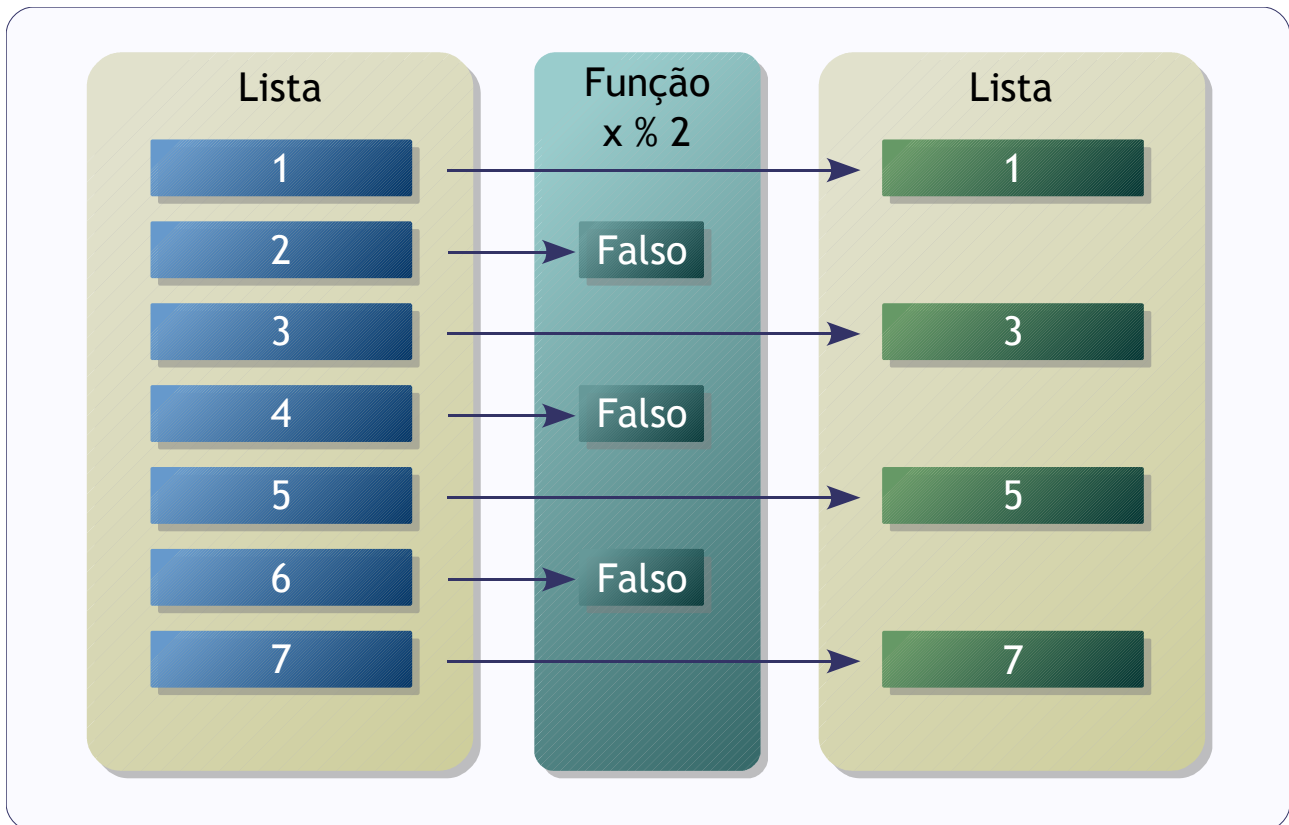
O mapeamento consiste em aplicar uma função a todos os itens de uma seqüência, gerando outra lista contendo os resultados e com o mesmo tamanho da lista inicial.

No Python, o mapeamento é implementado pela função *map()*.

Exemplos:

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# log na base 10
from math import log10
print map(log10, nums)

# Dividindo por 3
print map(lambda x: x / 3, nums)
```

A função *map()* sempre retorna uma lista.

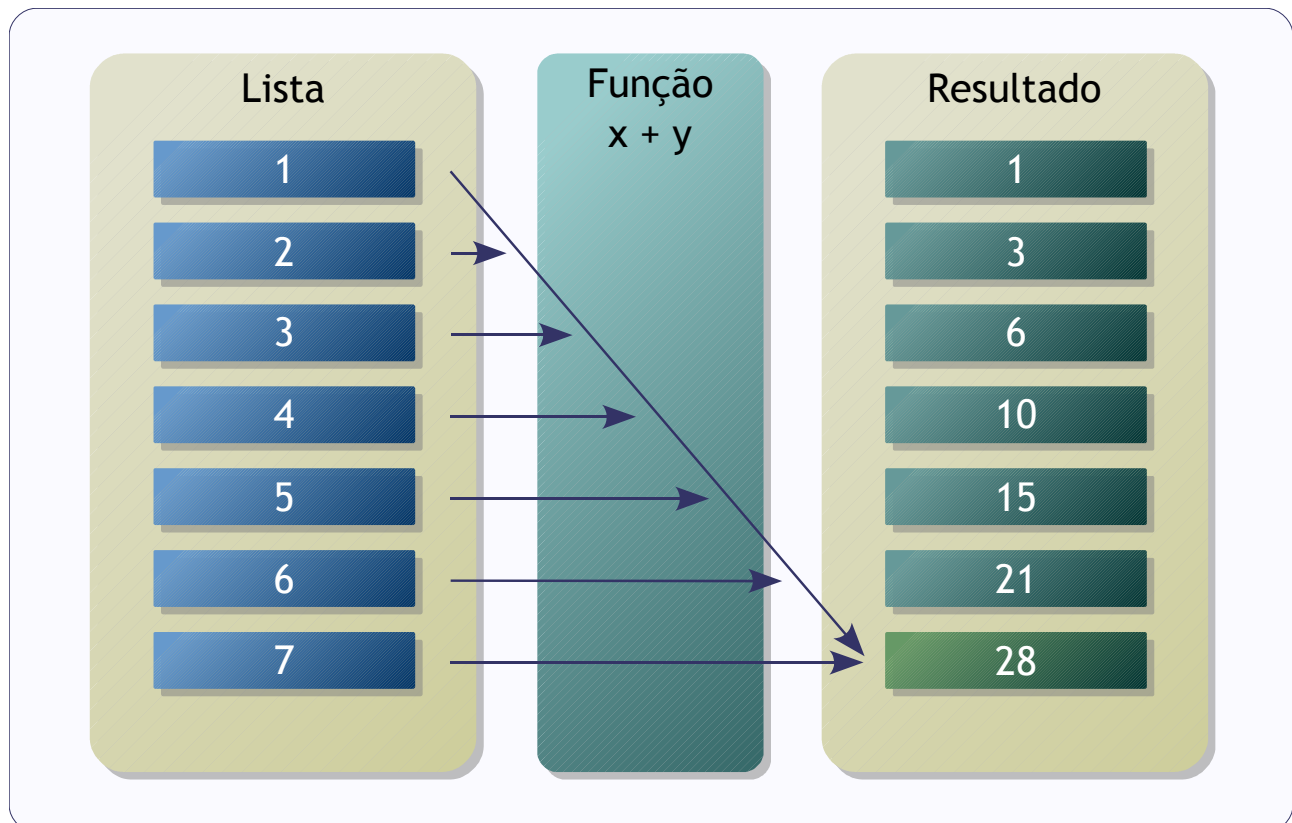
Filtragem

Na filtragem, uma função é aplicada em todos os itens de uma seqüência, se a função retornar um valor que seja avaliado como verdadeiro, o item original fará parte da seqüência resultante.

No Python, a filtragem é implementada pela função *filter()*.
Exemplo:

```
# Selecionando apenas os ímpares
print filter(lambda x: x % 2, nums)
```

A função *filter()* aceita também funções *lambda*, além de funções convencionais.

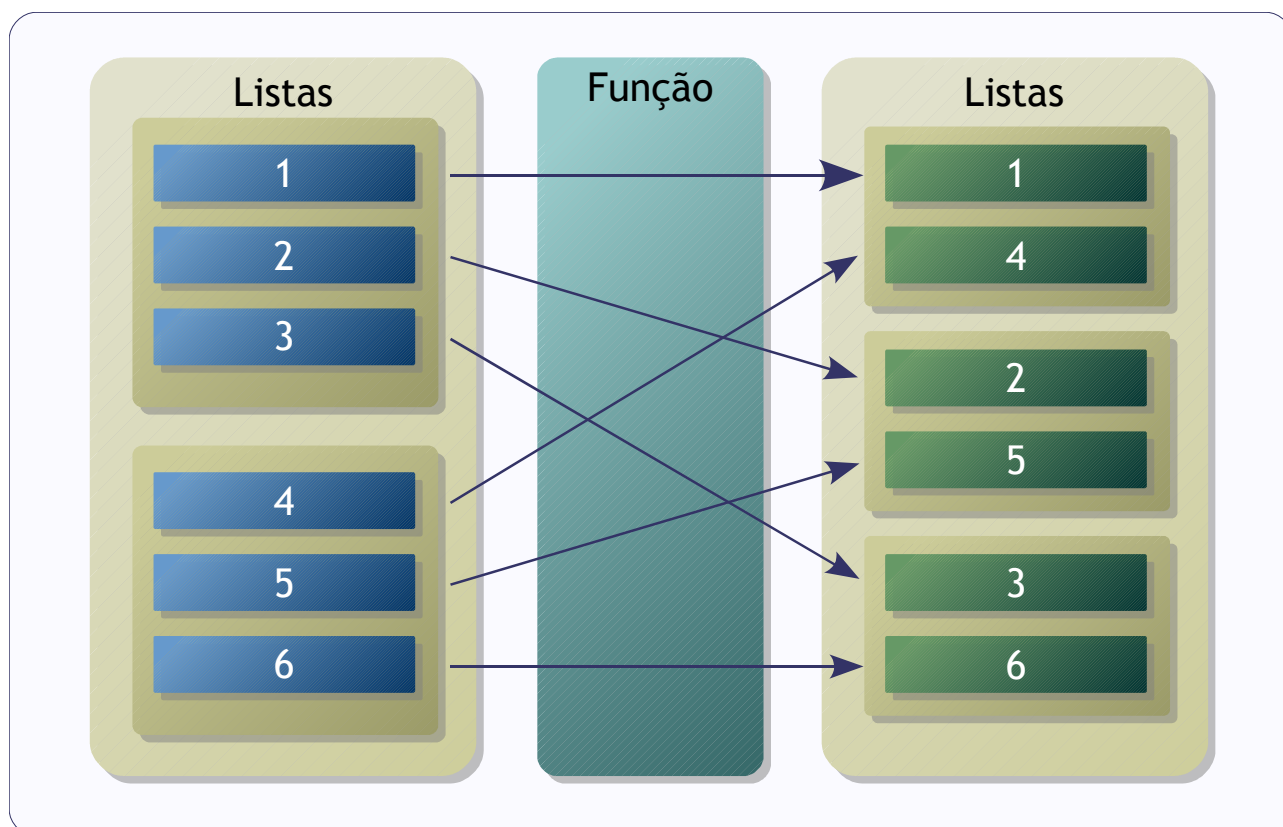
Redução

Redução significa aplicar uma função que recebe dois parâmetros, nos dois primeiros elementos de uma seqüência, aplicar novamente a função usando como parâmetros o resultado do primeiro par e o terceiro elemento, seguindo assim até o final da seqüência. O resultado final da redução é apenas um elemento.

Exemplos de redução, que é implementada no Python pela função *reduce()*:

```
# -*- coding: latin1 -*-
nums = range(100)

# Soma com reduce (pode concatenar strings)
print reduce(lambda x, y: x + y, nums)

# Soma mais simples, mas só para números
print sum(nums)

# Multiplicação
print reduce(lambda x, y: x * y, nums)
```

A função *sum()* é mais eficiente que *reduce()*.

Transposição

Transposição é construir uma série de seqüências a partir de outra série de seqüências, aonde a primeira nova seqüência contém o primeiro elemento de cada seqüência original, a segunda nova seqüência contém o segundo elemento de cada seqüência original, até que alguma das seqüências originais acabe.

Exemplo de transposição, que é implementada no Python pela função *zip()*:

```
# Uma lista com ('a', 1), ('b', 2), ...
from string import ascii_lowercase
print zip(ascii_lowercase, range(1, 100))

# Transposta de uma matriz
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print zip(*matriz)
```

A função *zip()* sempre retorna uma lista de tuplas.

List Comprehension

Em computação, *List Comprehension* é uma construção equivalente a notações matemáticas do tipo:

$$S=\{x^2 \forall\ x\ em\ \mathbb{N},\ x \geqslant 20\}$$

Ou seja, S é o conjunto formado por x ao quadrado para todo x no conjunto dos números naturais, se x for maior ou igual a 20.

Sintaxe:

```
lista = [ <expressão> for <referência> in <seqüência> if <condição> ]
```

Exemplo:

```
# -*- coding: latin1 -*-

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

# Eleve os ímpares ao quadrado
print [ x**2 for x in nums if x % 2 ]
```

Saída:

```
[1, 9, 25, 49, 81, 121]
```

O *List Comprehension* é mais eficiente do que usar as funções *map()* e *filter()*.

Generator Expression

Generator Expression é uma expressão semelhante ao *List Comprehension* que funciona como um gerador.

Exemplo:

```
# -*- coding: latin1 -*-

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
# Eleve os ímpares ao quadrado
gen = ( x**2 for x in nums if x % 2 )

# Mostra os resultados
for num in gen:
   print num
```

O *Generator Expression* usa menos recursos do que o *List Comprehension* equivalente.

Exercícios III

1. Implementar um gerador de números primos.

2. Implementar o gerador de números primos como uma expressão (dica: use o módulo *itertools*).

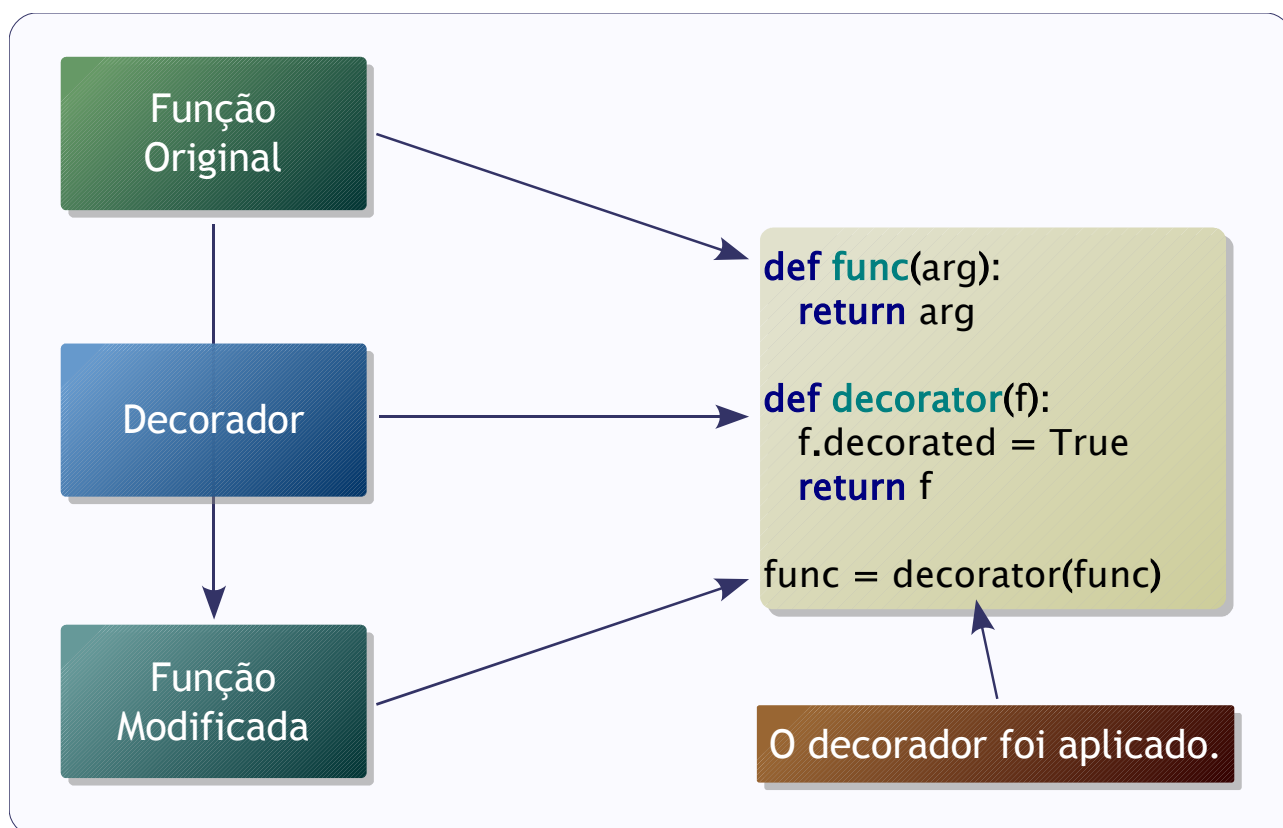
3. Implementar um gerador que produza tuplas com as cores do padrão RGB (R, G e B variam de 0 a 255) usando *xrange()* e uma função que produza uma lista com as tuplas RGB usando *range()*. Compare a performance.

4. Implementar um gerador que leia um arquivo e retorne uma lista de tuplas com os dados (o separador de campo do arquivo é vírgula), eliminando as linhas vazias. Caso ocorra algum problema, imprima uma mensagem de aviso e encerre o programa.

Parte IV

- Decoradores.
- Classes.
- Testes automatizados.
- Exercícios IV.

Decoradores

Decoradores (*decorators*) são funções que são aplicadas em outras funções e retornam funções modificadas. Decoradores tanto podem ser usados para alterar propriedades das funções (que são objetos) quanto para “envolver” as funções, acrescentando uma camada em torno delas com novas funcionalidades.

A partir do Python 2.4, o caractere “@” pode ser usado para automatizar o processo de aplicar o decorador:

```
def decorator(f):
  f.decorated = True
  return f

@decorator
def func(arg):
  return arg
```

Com isso, foi criado um atributo novo na função, que pode ser usado depois, quando a função for executada.

Exemplo:

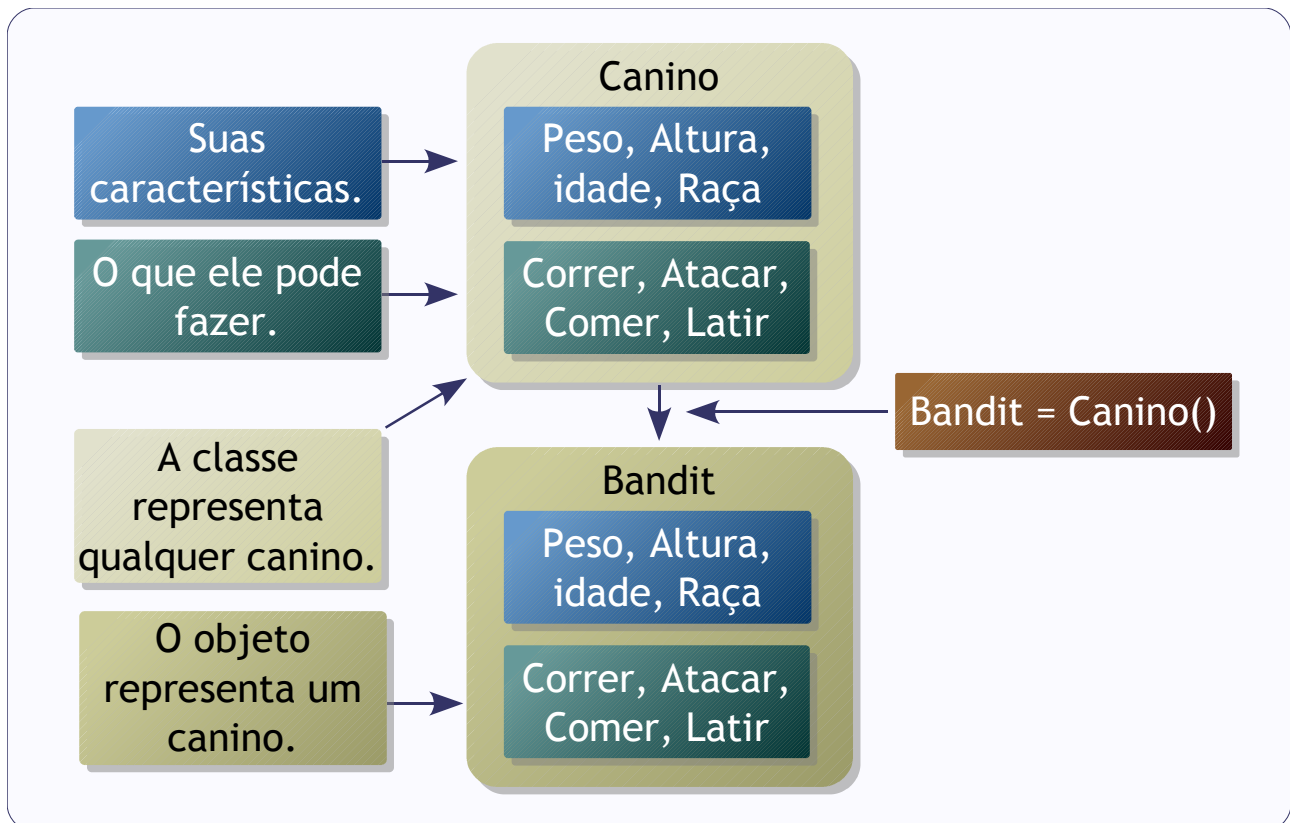
```
# -*- coding: latin1 -*-

# Função decoradora
def dumpargs(f):

    # Função que envolverá a outra
    def func(*args):

        # Mostra os argumentos passados para a função
        print args

        # Retorna o resultado da função original
        return f(*args)

    # Retorna a função modificada
    return func

@dumpargs
def multiply(*nums):

    m = 1

    for n in nums:
        m = m * n
    return m

print multiply(1, 2, 3)
```

Saída:

```
(1, 2, 3)
6
```

A saída apresenta os parâmetros que a função decorada recebeu.

Classes

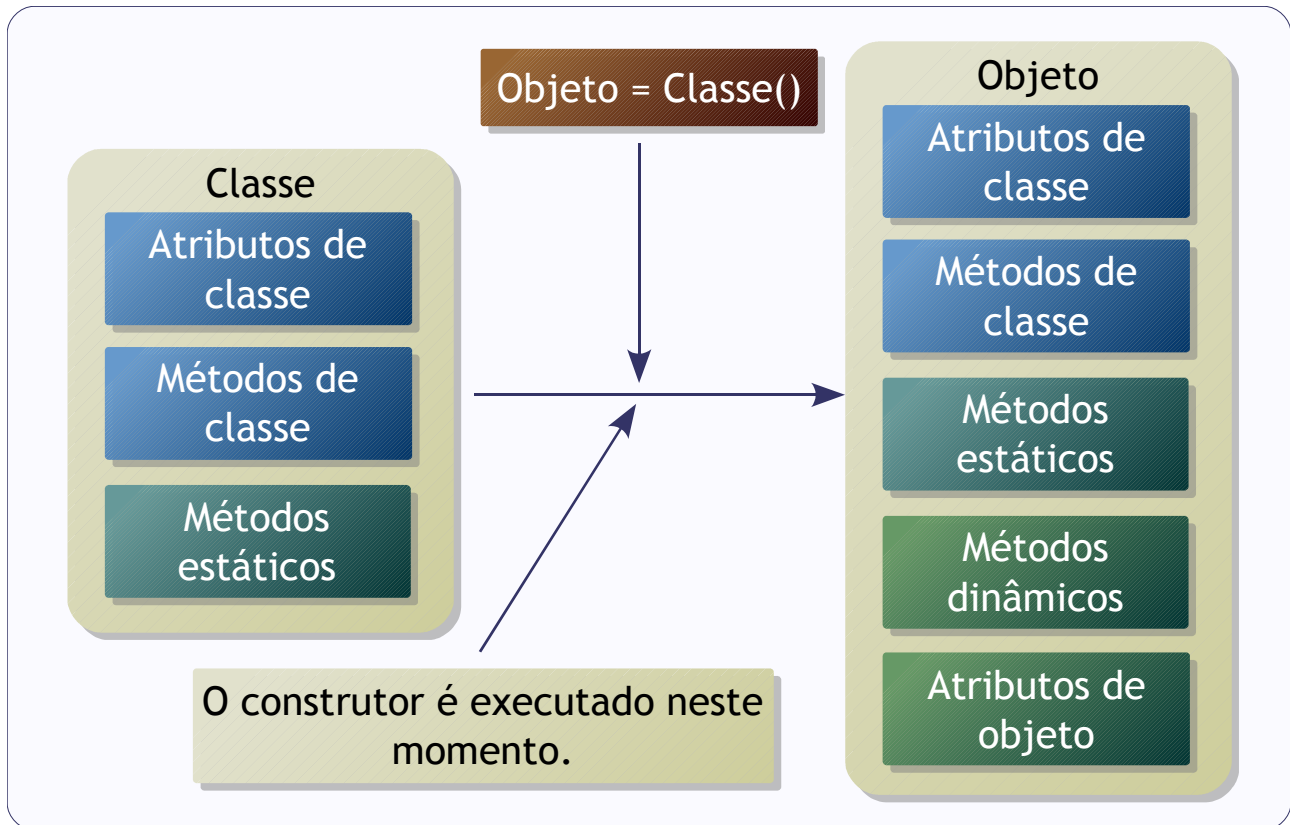
Um objeto é uma abstração computacional que representa uma entidade, com qualidades (atributos) e ações (métodos) que a entidade pode realizar. A classe é a estrutura básica do paradigma de orientação a objetos. A classe representa o tipo do objeto, que define as qualidades do objeto e o que ele pode fazer.

Por exemplo, a classe *Canino* descreve as características e ações dos caninos em geral, enquanto o objeto *Bandit* representa um canino em especial.

Os atributos são estruturas de dados sobre o objeto e os métodos são funções que descrevem como o objeto se comporta.

No Python, novos objetos são criados a partir das classes através de atribuição.

Quando um novo objeto é criado, o construtor da classe é executado. Em Python, o construtor é um método especial chamado *__new__()*. Após a chamada ao construtor, o método *__init__()* é chamado para inicializar a nova instância.

Um objeto continua existindo na memória enquanto existir pelo menos uma referência a

ele. O interpretador Python possui um recurso chamado coletor de lixo (*Garbage Collector*) que limpa da memória objetos sem referências[17]. Quando o objeto é apagado, o método especial *__done__()* é evocado. Funções ligadas ao coletor de lixo podem ser encontradas no módulo *gc*.

Em Python:

- Quase tudo é objeto, mesmo os tipos básicos, como números inteiros.
- Tipos e classes são unificados.
- Os operadores são na verdade chamadas para métodos especiais.
- As classes são abertas (menos para os tipos *builtins*).

Métodos especiais são identificados por nomes no padrão *__metodo__()* (dois sublinhados no início e no final do nome) e definem como os objetos derivados da classe se comportarão em situações particulares, como em sobrecarga de operadores.

As classes *new style* são derivadas da classe *object* e podem utilizar recursos novos das classes do Python, como *properties* e *metaclasses*. As *properties* são atributos calculados em tempo de execução através de métodos, enquanto as *metaclasses* são classes que geram classes, com isso permitem personalizar o comportamento das classes.

17 Para apagar uma referência a um objeto, use o comando *del*. Se todas as referências forem apagadas, o *Garbage Collector* apagará o objeto.

Sintaxe:

```
# -*- coding: latin1 -*-

class Classe(supcl1, supcl2):
    """
    Isto é uma classe
    """
    clsvar = []

    def __init__(self, args):
        """
        Inicializador da classe
        """
        <bloco de código>

    def __done__(self):
        """
        Destrutor da classe
        """
        <bloco de código>

    def metodo(self, params):
        """
        Método de objeto
        """
        <bloco de código>

    @classmethod
    def cls_metodo(cls, params):
        """
        Método de classe
        """
        <bloco de código>

    @staticmethod
    def est_metodo(params):
        """
        Método estático
        """
        <bloco de código>


obj = Classe()
obj.metodo()

Classe.cls_metodo()
Classe.est_metodo()
```

Métodos estáticos são aqueles que não tem ligação com atributos do objeto ou da classe.

Funcionam como as funções comuns.

Métodos de classe podem usar apenas atributos e outros métodos de classe. O argumento *cls* representa a classe em si, precisa ser passado explicitamente como primeiro parâmetro do método.

Métodos de objeto podem usar atributos e outros métodos do objeto. A variável *self*, que representa o objeto e também precisa ser passado de forma explícita. O nome *self* é uma convenção, assim como *cls*, podendo ser trocado por outro nome qualquer, porém é considerada como boa prática manter o nome.

Exemplo de classe:

```
# -*- coding: latin1 -*-


class Cell(object):
    """
    Classe para células de planilha
    """

    def __init__(self, formula='""', format='%s'):
        """
        Inicializa a célula
        """

        self.formula = formula
        self.format = format

    def __repr__(self):
        """
        Retorna a representação em string da célula
        """

        return self.format % eval(self.formula)


print Cell('123**2')
print Cell('23*2+2')
print Cell('abs(-1.45 / 0.3)', '%2.3f')
```

Saída:

```
15129
48
4.833
```

O método *__repr__()* é usado internamente pelo comando *print* para obter uma representação do objeto em forma de texto.

Classes abertas

No Python, as classes que não são *builtins* podem ser alteradas em tempo de execução, devido a natureza dinâmica da linguagem. É possível acrescentar métodos e atributos novos, por exemplo. A mesma lógica se aplica aos objetos.

Exemplo de como acrescentar um novo método:

```
# -*- coding: latin1 -*-


class User(object):
    """Uma classe bem simples.
    """
    def __init__(self, name):
        """Inicializa a classe, atribuindo um nome
        """
        self.name = name


# Um novo método para a classe
def set_password(self, password):
    """Troca a senha
    """
    self.password = password

print 'Classe original:', dir(User)

# O novo método é inserido na classe
User.set_password = set_password
print 'Classe modificada:', dir(User)

user = User('guest')
user.set_password('guest')

print 'Objeto:', dir(user)
print 'Senha:', user.password
```

Saída:

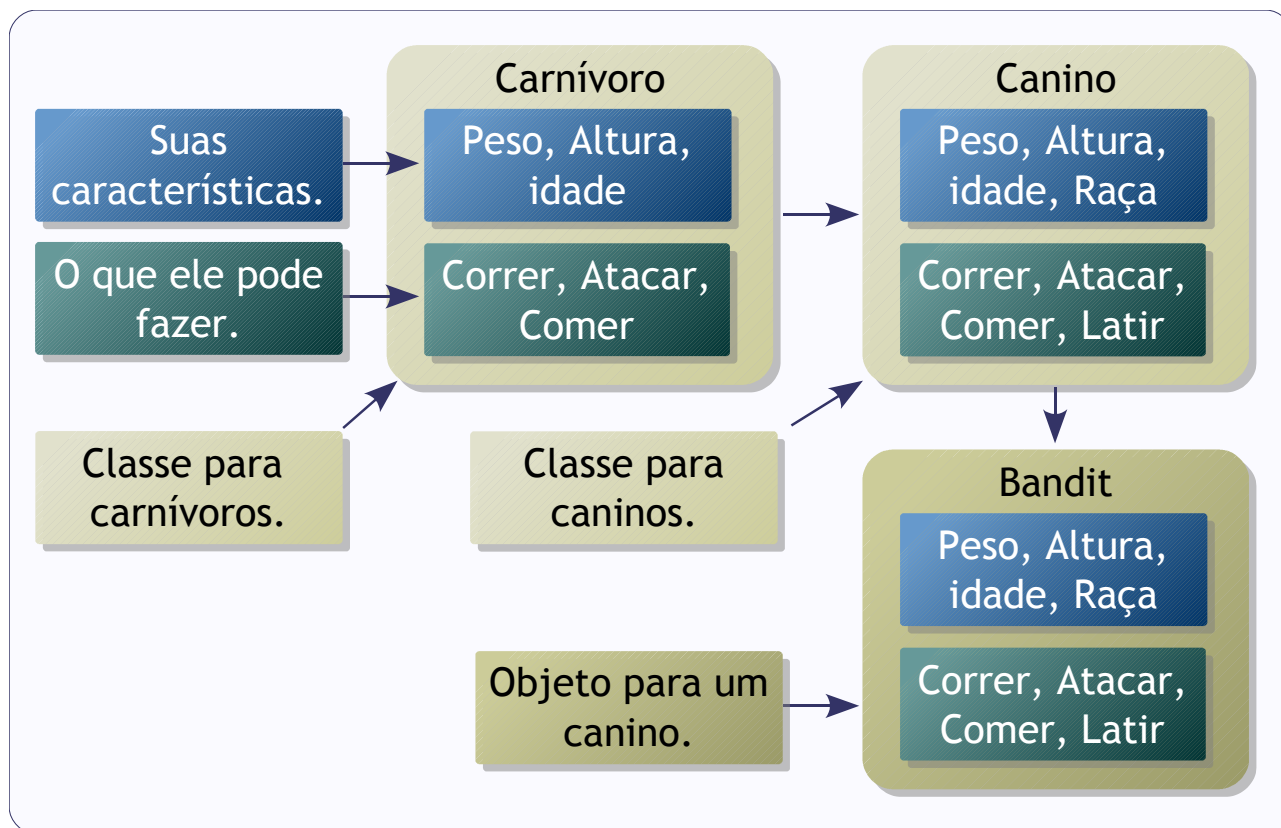
```
Classe original: ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', '__weakref__']
Classe modificada: ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'set_password']
Objeto: ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
```

```
'__setattr__', '__str__', '__weakref__', 'name', 'password', 'set_password']
Senha: guest
```

A classe modificada passou a ter um novo método: *set_password()*.

Acrescentar métodos ou atributos em classes abertas é mais simples do que criar uma nova classe através de herança com os novos métodos ou atributos.

Herança simples

Herança é um mecanismo que a orientação a objeto provê, com objetivo de facilitar o reaproveitamento de código. A idéia é que as classes sejam construídas formando uma hierarquia.

A nova classe pode implementar novos métodos e atributos e herdar métodos e atributos da classe antiga (que também pode ter herdado de classes anteriores), porém estes métodos e atributos podem substituídos no processo. Na herança simples, a classe deriva de somente uma classe já existente.

Exemplo de herança simples:

```
class Pendrive(object):

    def __init__(self, tamanho, interface='2.0'):

        self.tamanho = tamanho
        self.interface = interface
```

```
class MP3Player(Pendrive):

    def __init__(self, tamanho, interface='2.0', turner=False):

        self.turner = turner
        Pendrive.__init__(self, tamanho, interface)


mp3 = MP3Player(1024)

print '%s\n%s\n%s' % (mp3.tamanho, mp3.interface, mp3.turner)
```

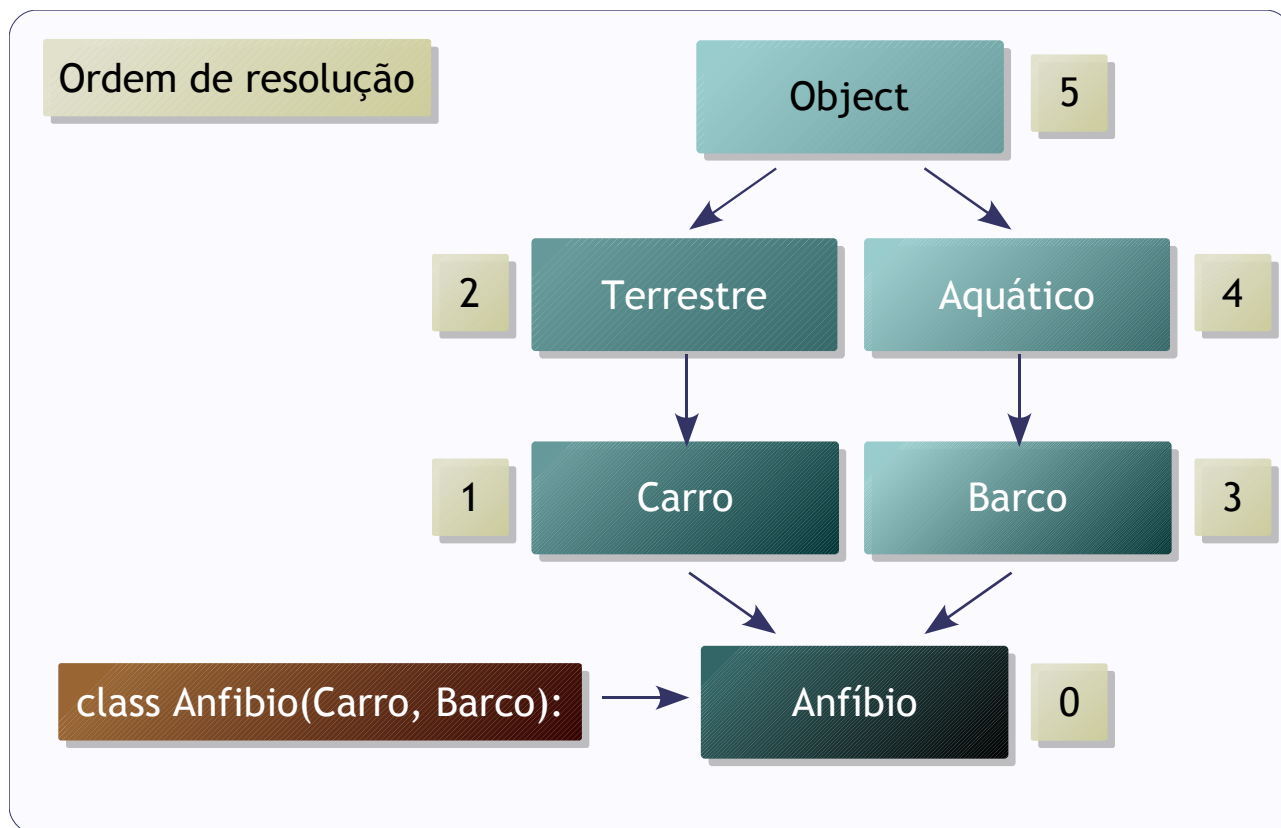
A classe MP3Player é derivada da classe Pendrive.

Saída:

```
1024
2.0
False
```

A classe *MP3Player* herda de *Pendrive* o tamanho e a interface.

Herança múltipla

Na herança múltipla, a nova classe deriva de várias classes já existentes. A diferença mais significativa em relação à herança simples é a ordem de resolução de métodos (em inglês, *Method Resolution Order*- MRO), que segue o chamado algoritmo diamante.

No algoritmo diamante, a resolução será feita a partir da esquerda, descendo até encontrar a classe em comum entre os caminhos dentro hierarquia. Quando encontra a classe em comum, passa para o caminho à direita. Ao esgotar os caminhos, o algoritmo prossegue para a classe em comum e repete o processo.

Exemplo:

```
# -*- coding: latin1 -*-

class Terrestre(object):
    """
    Classe de veículos terrestres
    """
    se_move_em_terra = True
```

```
    def __init__(self, velocidade=100):
        """
        Inicializa o objeto
        """
        self.velocidade_em_terra = velocidade


class Aquatico(object):
    """
    Classe de veículos aquaticos
    """
    se_move_na_agua = True

    def __init__(self, velocidade=5):
        """
        Inicializa o objeto
        """
        self.velocidade_agua = velocidade


class Carro(Terrestre):
    """
    Classe de carros
    """
    rodas = 4

    def __init__(self, velocidade=120, pistoes=4):
        """
        Inicializa o objeto
        """
        self.pistoes = pistoes
        Terrestre.__init__(self, velocidade=velocidade)


class Barco(Aquatico):
    """
    Classe de barcos
    """
    def __init__(self, velocidade=6, helices=1):
        """
        Inicializa o objeto
        """
        self.helices = helices
        Aquatico.__init__(self, velocidade=velocidade)


class Anfibio(Carro, Barco):
    """
    Classe de anfíbios
    """
    def __init__(self, velocidade_em_terra=80,
```

A classe *Carro* deriva de *Terrestre*.

A classe *Barco* deriva de *Aquatico*.

A classe *Anfibio* é derivada de *Carro* e *Barco*.

```
        velocidade_na_agua=4, pistoes=6, helices=2):
        """
        Inicializa o objeto
        """
        # É preciso evocar o __init__ de cada classe pai
        Carro.__init__(self, velocidade=velocidade_em_terra,
            pistoes=pistoes)
        Barco.__init__(self, velocidade=velocidade_na_agua,
            helices=helices)


novo_anfibio = Anfibio()

for atr in dir(novo_anfibio):

    # Se não for método especial:
    if not atr.startswith('__'):
        print atr, '=', getattr(novo_anfibio, atr)
```

Saída:

```
helices = 2
pistoes = 6
rodas = 4
se_move_em_terra = True
se_move_na_agua = True
velocidade_agua = 4
velocidade_em_terra = 80
```

Na hierarquia de classes do exemplo, a MRO para a classe dos anfíbios será:

```
[<class '__main__.Anfibio'>,
<class '__main__.Carro'>,
<class '__main__.Terrestre'>,
<class '__main__.Barco'>,
<class '__main__.Aquatico'>,
<type 'object'>]
```

A herança múltipla é um recurso que gera muita controvérsia, pois seu uso pode tornar o projeto confuso e obscuro.

Propriedades

Propriedades (*properties*) são atributos calculados em tempo de execução. As propriedades são criadas através da função *property*.

As vantagens de usar propriedades são:

- Validar a entrada do atributo.
- Criar atributos apenas de leitura.
- Facilitar o uso da classe[18].
- A facilidade de mudar de um atributo convencional para uma propriedade sem a necessidade de alterar as aplicações que utilizam a classe.

Exemplo de código sem propriedades:

```
# get_*, set_*...


class Projetil(object):

    def __init__(self, alcance, tempo):

        self.alcance = alcance
        self.tempo = tempo

    def get_velocidade(self):

        return self.alcance / self.tempo


moab = Projetil(alcance=10000, tempo=60)

print moab.get_velocidade()
```

Saída:

```
166
```

Exemplo de propriedade através de decorador:

```
# -*- coding: latin1 -*-
# Exemplo de Property de leitura
```

18 As propriedades escondem as funções *get()* e *set()* dos atributos, tornando o uso da classe mais simples.

```
class Projetil(object):

    def __init__(self, alcance, tempo):

        self.alcance = alcance
        self.tempo = tempo

    @property
    def velocidade(self):

        return self.alcance / self.tempo


moab = Projetil(alcance=10000, tempo=60)

# A velocidade é calculada
print moab.velocidade
```

Saída:

```
166
```

Exemplo de propriedade através de chamada de função:

```
# Property de leitura & escrita


class Projetil(object):

    def __init__(self, alcance, tempo):

        self.alcance = alcance
        self.tempo = tempo

    # Calcula a velocidade
    def getv(self):

        return self.alcance / self.tempo

    # Calcula o tempo
    def setv(self, v):

        self.tempo = self.alcance / v

    # Define a propriedade
    velocidade = property(getv, setv)
```

```
moab = Projetil(alcance=10000, tempo=60)
print moab.velocidade

# Muda a velocidade
moab.velocidade = 350
print moab.tempo
```

Saída:

```
166
28
```

Propriedades são particularmente interessantes para quem desenvolve bibliotecas para serem usadas por outras pessoas.

Sobrecarga de operadores

No Python, o comportamento dos operadores é definido por métodos especiais, porém tais métodos só podem ser alterados nas classes abertas. Por convenção, os métodos especiais têm nomes que começam e terminam com "__".

Exemplo:

```
# A classe String deriva de str
class String(str):

    def __sub__(self, s):

        return self.replace(s, '')

s1 = String('The Lamb Lies Down On Broadway')
s2 = 'Down '

print '"%s" - "%s" = "%s"' % (s1, s2, s1 - s2)
```

Saída:

```
"The Lamb Lies Down On Broadway" - "Down " = "The Lamb Lies On Broadway"
```

Lista de operadores e os métodos correspondentes:

Operador	Método	Operação
+	__add__	adição
-	__sub__	subtração
*	__mul__	multiplicação
**	__pow__	potência
/	__div__	divisão
//	__floordiv__	divisão truncada
%	__mod__	módulo
+	__pos__	positivo
-	__neg__	negativo

Operador	Método	Operação
<	__lt__	menor que
>	__gt__	maior que
<=	__le__	menor ou igual a
>=	__ge__	maior ou igual a
==	__eq__	Igual a
!=	__ne__	diferente de
<<	__lshift__	deslocamento para esquerda
>>	__rshift__	deslocamento para direita
&	__and__	e bit-a-bit
|	__or__	ou bit-a-bit
^	__xor__	ou exclusivo bit-a-bit
~	__inv__	inversão

Observações:

- A subtração definida no código não é comutativa (da mesma forma que a adição em *strings* também não é)
- A classe *str* não é aberta, portanto não é possível alterar o comportamento da *string* padrão do Python. Porém a classe *String* é aberta.
- A redefinição de operadores conhecidos pode dificultar a leitura do código.

Metaclasses

Em uma linguagem orientada a objeto aonde (quase) tudo são objetos e todo o objeto tem uma classe, é natural que as classes também podem ser tratadas como objetos.

Metaclasse é uma classe cujas as instâncias são classes, sendo assim, a metaclasse define o comportamento das outras classes. Em Python, a classe *type* é uma metaclasse e pode ser usada para criar novas metaclasses.

Exemplo de metaclasse criada a partir de *type*:

```
# -*- coding: latin1 -*-


class Singleton(type):
    """
    Metaclasse Singleton
    """

    def __init__(cls, name, bases, dic):

        type.__init__(cls, name, bases, dic)

        # Retorna o próprio objeto na cópia
        def __copy__(self):
            return self

        # Retorna o próprio objeto na cópia recursiva
        def __deepcopy__(self, memo=None):
            return self

        cls.__copy__ = __copy__
        cls.__deepcopy__ = __deepcopy__

    def __call__(cls, *args, **kwargs):

        # Chamada que cria novos objetos,
        # aqui retorna sempre o mesmo
        try:
            return cls.__instance

        # Se __instance não existir, então crie...
        except AttributeError:

            # A função super() pesquisa na MRO
            # a partir de Singleton
            cls.__instance = super(Singleton,
                cls).__call__(*args, **kwargs)
```

```
        return cls.__instance

import MySQLdb

class Con(object):
    """
    Classe de conexão única
    """

    # Define a metaclasse desta classe
    __metaclass__ = Singleton

    def __init__(self):

        # Cria uma conexão e um cursor
        con = MySQLdb.connect(user='root')
        self.db = con.cursor()
        # Sempre será usado o mesmo
        # objeto de cursor


class Log(object):
    """
    Classe de log
    """

    # Define a metaclasse desta classe
    __metaclass__ = Singleton

    def __init__(self):

        # Abre o arquivo de log para escrita
        self.log = file('msg.log', 'w')
        # Sempre será usado o mesmo
        # objeto de arquivo

    def write(self, msg):

        print msg
        # Acrescenta as mensagens no arquivo
        self.log.write(str(msg) + '\n')


# Conexão 1
con1 = Con()
Log().write('con1 id = %d' % id(con1))
con1.db.execute('show processlist')
Log().write(con1.db.fetchall())

# Conexão 2
```

```
con2 = Con()
Log().write('con2 id = %d' % id(con2))
con2.db.execute('show processlist')
Log().write(con2.db.fetchall())

import copy

# Conexão 3
con3 = copy.copy(con1)
Log().write('con3 id = %d' % id(con3))
con3.db.execute('show processlist')
Log().write(con2.db.fetchall())
```

Saída e conteúdo do arquivo "msg.log":

```
con1 id = 10321264
((20L, 'root', 'localhost:1125', None, 'Query', 0L, None, 'show processlist'),)
con2 id = 10321264
((20L, 'root', 'localhost:1125', None, 'Query', 0L, None, 'show processlist'),)
con3 id = 10321264
((20L, 'root', 'localhost:1125', None, 'Query', 0L, None, 'show processlist'),)
```

Com isso, todas as referências apontam para o mesmo objeto.

Testes automatizados

Testar software é uma tarefa repetitiva, demorada e tediosa. Por isso, surgiram várias ferramentas para automatizar testes. Existem dois módulos para testes automatizados que acompanham o Python: *doctest* e *unittest*.

O módulo doctest usa *Doc Strings* presentes no código para definir os testes do código. O *doctest* procura por um trecho de texto seja semelhante a uma sessão interativa de Python, executa a mesma seqüência de comandos, analisa a saída e faz um relatório dos testes que falharam, com os erros encontrados.

Exemplo:

```
"""
fib.py

Implementa Fibonacci.
"""

def fib(n):
    """Fibonacci:
    Se n <= 1, fib(n) = 1
    Se n > 1, fib(n) = fib(n - 1) + fib(n - 2)

    Exemplos de uso:

    >>> fib(0)
    1
    >>> fib(1)
    1
    >>> fib(10)
    89
    >>> [ fib(x) for x in xrange(10) ]
    [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
    >>> fib('')
    Traceback (most recent call last):
      File "<input>", line 1, in ?
      File "<input>", line 19, in fib
    TypeError
    >>>
    """
    if not type(n) is int:
        raise TypeError

    if n > 1:
        return fib(n - 1) + fib(n - 2)
    else:
        return 1
```

Testes para o *doctest*.

```
def _doctest():
    """
    Evoca o doctest.
    """

    import doctest
    doctest.testmod()

if __name__ == "__main__":

    _doctest()
```

Os testes serão executados se este módulo for evocado diretamente pelo Python.

Se todos os testes forem bem sucedidos, não haverá relatório dos testes.

Exemplo de relatório de erros do testes (a *Doc String* foi alterada de propósito para gerar um erro):

```
**********************************************************************
File "fib.py", line 18, in __main__.fib
Failed example:
    fib(10)
Expected:
    89
Got:
    100
**********************************************************************
1 items had failures:
   1 of   5 in __main__.fib
***Test Failed*** 1 failures.
```

No *unittest*, os testes são criados através de uma subclasse da classe *unittest.TestCase*. Os testes são definidos como métodos da subclasse. Os métodos devem ter seus nomes iniciando com "test".

Os métodos de teste devem evocar ao terminar um dos métodos:

- *assert_*: verifica se uma condição é atingida.
- *assertEqual*: verifica se o resultado é igual ao parâmetro passado.
- *AssertRaises*: verifica se a exceção é a esperada.

Se houver um método chamado *setUp*, este será executado antes de cada teste, assim é possível reinicializar variáveis e garantir que um teste não prejudique o outro. O final dos testes, o *unittest* gera o relatório dos testes.

Exemplo:

```
"""
fibtest.py

Usa unittest para testar fib.py.
"""

import fib
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = range(10)

    def test0(self):
        self.assertEqual(fib.fib(0), 1)

    def test1(self):
        self.assertEqual(fib.fib(1), 1)

    def test10(self):
        self.assertEqual(fib.fib(10), 89)

    def testseq(self):
        fibs = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

        for x, y in zip(fibs, [ fib.fib(x) for x in self.seq ]):
            self.assert_(x is y)

    def testtype(self):
        self.assertRaises(TypeError, fib.fib, '')

if __name__ == '__main__':

    unittest.main()
```

Métodos que definem os testes.

Saída:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Exemplo de relatório com erros:

```
..F..
```

```
=================================================================
=============
FAIL: test10 (__main__.TestSequenceFunctions)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittest1.py", line 22, in test10
    self.assertEqual(fib.fib(10), 89)
AssertionError: 100 != 89

----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (failures=1)
```

No relatório, o terceiro teste falhou, pois “fib.fib(10)” retornou 100 ao invés de 89, como seria o esperado.

O *unittest* oferece uma solução muito semelhante a bibliotecas de testes implementadas em outras linguagens, enquanto o *doctest* é mais simples de usar e se integra bem com a documentação (as sessões do *doctest* podem servir como exemplos de uso).

Exercícios IV

1. Crie uma classe que modele um quadrado, com um atributo lado e os métodos: mudar valor do lado, retornar valor do lado e calcular área.

2. Crie uma classe derivada de lista com um método retorne os elementos da lista sem repetição.

3. Implemente uma classe *Carro* com as seguintes propriedades:
    - Um veículo tem um certo consumo de combustível (medidos em km / litro) e uma certa quantidade de combustível no tanque.
    - O consumo é especificado no construtor e o nível de combustível inicial é 0.
    - Forneça um método *mover(km)* que receba a distância em quilômetros e reduza o nível de combustível no tanque de gasolina.
    - Forneça um método *gasolina()*, que retorna o nível atual de combustível.
    - Forneça um método *abastecer(litros)*, para abastecer o tanque.

4. Implementar uma classe *Vetor*:
    - Com coordenadas x, y e z.
    - Que suporte soma, subtração, produto escalar, produto vetorial.
    - Que calcule o módulo (valor absoluto) do vetor.

5. Implemente um módulo com:
    - Uma classe *Ponto*, com coordenadas x, y e z.
    - Uma classe *Linha*, com dois pontos A e B, e que calcule o comprimento da linha.
    - Uma classe *Triangulo*, com dois pontos A, B e C, que calcule o comprimento dos lados e a área.

Parte V

NumPy

NumPy[19] é um pacote que inclui:

- Classe *array*.
- Classe *matrix*.
- Várias funções auxiliares.

Arranjos

A classe *array* implementa um arranjo homogêneo mutável com número arbitrário de elementos, semelhante à lista comum do Python, porém mais poderosa.

```
import numpy

# Criando arranjos

print 'Arranjo criado a partir de uma lista:'
a = numpy.array([0, 1, 2, 3, 4, 5, 6, 7, 8])

print a
# [0 1 2 3 4 5 6 7 8]

print 'Arranjo criado a partir de um intervalo:'
z = numpy.arange(0., 4.5, .5)

print z
# [ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4. ]

print 'Arranjo de 1s 2x3:'
y = numpy.ones((2, 3))

print y
# [[ 1.  1.  1.]
#  [ 1.  1.  1.]]

print 'Arranjos podem gerar novos arranjos:'
# numpy.round() é uma função do numpy
# semelhante ao builtin round(), porém aceita
# arranjos como parâmetro
cos = numpy.round(numpy.cos(z), 1)

print cos
# [ 1.   0.9  0.5  0.1 -0.4 -0.8 -1.  -0.9 -0.7]

print 'Multiplicando cada elemento por um escalar:'
print 5 * z
# [  0.    2.5   5.    7.5  10.   12.5  15.   17.5  20. ]
```

19 Os fontes, binários e documentação podem ser encontrados em: http://numpy.scipy.org/.

```
print 'Somando arranjos elemento por elemento:'
print z + cos
# [ 1.   1.4  1.5  1.6  1.6  1.7  2.   2.6  3.3]

print 'Redimensionando o arranjo:'
z.shape = 3, 3

print z
# [[ 0.   0.5  1. ]
#  [ 1.5  2.   2.5]
#  [ 3.   3.5  4. ]]

print 'Arranjo transposto:'
print z.transpose()
# [[ 0.   1.5  3. ]
#  [ 0.5  2.   3.5]
#  [ 1.   2.5  4. ]]

print '"Achata" o arranjo:'
print z.flatten()
# [ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4. ]

print 'O acesso aos elementos funciona como nas listas:'
print z[1]
# [ 1.5  2.   2.5]

print 'Caso especial, diferente da lista:'
print z[1, 1]
# 2.0

# Dados sobre o arranjo

print 'Formato do arranjo:'
print z.shape
# (3, 3)

print 'Quantidade de eixos:'
print z.ndim
# 2

print 'Tipo dos dados:'
print z.dtype
# float64
```

Saída completa:

```
Arranjo criado a partir de uma lista:
[0 1 2 3 4 5 6 7 8]
Arranjo criado a partir de um intervalo:
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4. ]
Arranjo de 1s 2x3:
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
Arranjos podem gerar novos arranjos:
[ 1.   0.9  0.5  0.1 -0.4 -0.8 -1.  -0.9 -0.7]
Multiplicando cada elemento por um escalar:
[  0.    2.5   5.    7.5  10.   12.5  15.   17.5  20. ]
Somando arranjos elemento por elemento:
[ 1.   1.4  1.5  1.6  1.6  1.7  2.   2.6  3.3]
Redimensionando o arranjo:
[[ 0.   0.5  1. ]
 [ 1.5  2.   2.5]
 [ 3.   3.5  4. ]]
Arranjo transposto:
[[ 0.   1.5  3. ]
 [ 0.5  2.   3.5]
 [ 1.   2.5  4. ]]
"Achata" o arranjo:
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4. ]
O acesso aos elementos funciona como nas listas:
[ 1.5  2.   2.5]
Caso especial, diferente da lista:
2.0
Formato do arranjo:
(3, 3)
Quantidade de eixos:
2
Tipo dos dados:
float64
```

Ao contrário da lista, os arranjos são homogêneos, ou seja, todos elementos são do mesmo tipo.

Matrizes

A classe *matrix* implementa operações de matrizes.

```
import numpy

print 'Criando uma matriz a partir de uma lista:'
l = [[3,4,5], [6, 7, 8], [9, 0, 1]]
Z = numpy.matrix(l)
print Z
# [[3 4 5]
#  [6 7 8]
#  [9 0 1]]

print 'Transposta da matriz:'
```

```
print Z.T
# [[3 6 9]
#  [4 7 0]
#  [5 8 1]]

print 'Inversa da matriz:'
print Z.I
# [[-0.23333333  0.13333333  0.1       ]
#  [-2.2         1.4        -0.2       ]
#  [ 2.1        -1.2         0.1       ]]

# Criando outra matriz
R = numpy.matrix([[3, 2, 1]])

print 'Multiplicando matrizes:'
print R * Z
# [[30 26 32]]

print 'Resolvendo um sistema linear:'
print numpy.linalg.solve(Z, numpy.array([0, 1, 2]))
# [ 0.33333333  1.         -1.        ]
```

Saída:

```
Criando uma matriz a partir de uma lista:
[[3 4 5]
 [6 7 8]
 [9 0 1]]
Transposta da matriz:
[[3 6 9]
 [4 7 0]
 [5 8 1]]
Inversa da matriz:
[[-0.23333333  0.13333333  0.1       ]
 [-2.2         1.4        -0.2       ]
 [ 2.1        -1.2         0.1       ]]
Multiplicando matrizes:
[[30 26 32]]
Resolvendo um sistema linear:
[ 0.33333333  1.         -1.        ]
```

O módulo numpy.linalg também implementa funções de decomposição de matrizes:

```
from numpy import *

# Matriz 3x3
A = array([(9, 4, 2), (5, 3, 1), (2, 0, 7)])
print 'Matriz A:'
```

```
print A

# Decompondo usando QR
Q, R = linalg.qr(A)

# Resultados
print 'Matriz Q:'
print Q
print 'Matriz R:'
print R

# Produto
print 'Q . R:'
print int0(dot(Q, R))
```

Saída:

```
Matriz A:
[[9 4 2]
 [5 3 1]
 [2 0 7]]
Matriz Q:
[[-0.85811633  0.14841033 -0.49153915]
 [-0.47673129 -0.58583024  0.65538554]
 [-0.19069252  0.79672913  0.57346234]]
Matriz R:
[[-10.48808848  -4.86265921  -3.52781158]
 [  0.          -1.16384941   5.28809431]
 [  0.           0.           3.68654364]]
Q . R:
[[9 4 2]
 [5 3 1]
 [2 0 7]]
```

O NumPy serve de base para diversos módulos, como o Matplolib, que implementa gráficos 2D e 3D, e o SciPy[20], que expande o NumPy com mais rotinas voltadas para a área cientifica.

20 Página oficial em: http://www.scipy.org/.

Gráficos

Existem vários pacotes de terceiros para a geração de gráficos disponíveis para Python, sendo que o mais popular deles é o Pylab / Matplotlib[21].

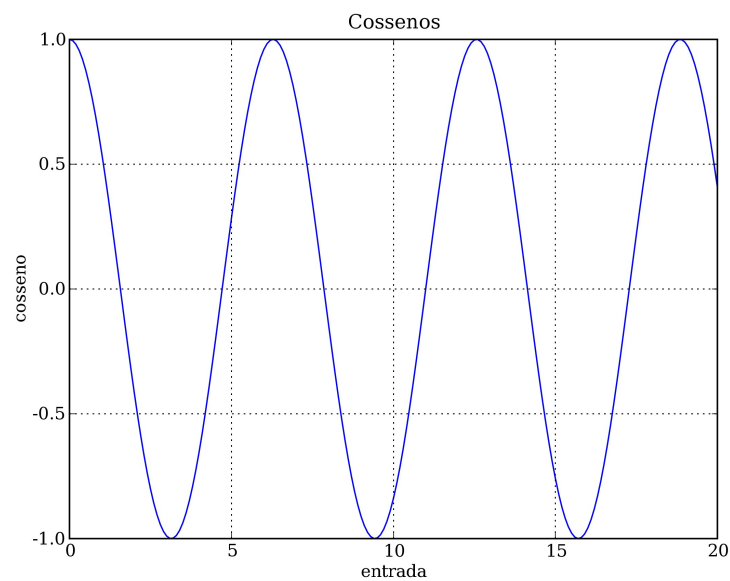
O pacote tem dois módulos principais:

- *matplotlib*: módulo que oferece uma abstração orientada a objetos aos recursos do pacote.
- *pylab*: módulo que oferece uma coleção de comandos semelhante ao Matlab, que é mais adequado para o uso interativo.

Exemplo:

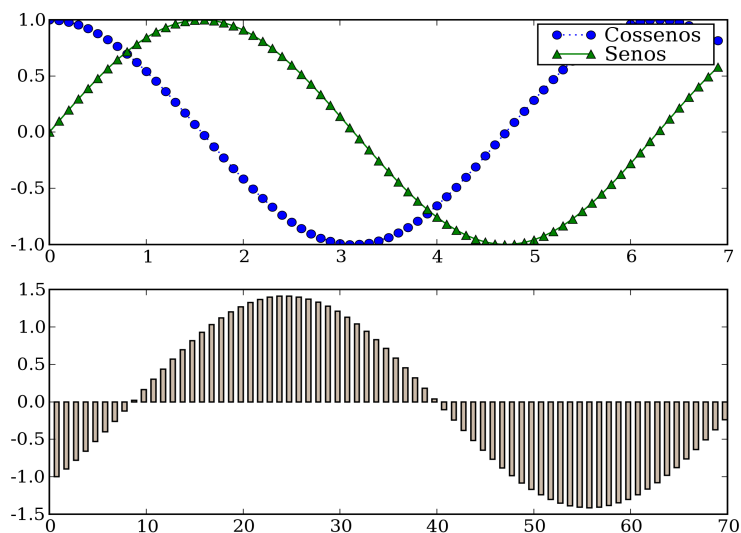
```
from pylab import *

ent = arange(0., 20.1, .1)

# Calcula os cossenos da entrada
sai = cos(ent)

# Plota a curva
plot(ent, sai)

# Texto para o eixo X
xlabel('entrada')

# Texto para o eixo Y
ylabel('cosseno')

# Texto no topo da figura
title('Cossenos')

# Ativa a grade
grid(True)

# Apresenta a figura resultante na tela
show()
```

Saída:

21 Os fontes, binários e documentação podem ser encontrados em: http://matplotlib.sourceforge.net/.

Outro exemplo:

```
from pylab import *

# Dados
ent1 = arange(0., 7., .1)
sai1 = cos(ent1)
sai2 = sin(ent1)
dif = sai2 - sai1

# Divide a figura em 2 linhas e 1 coluna,
# e seleciona a parte superior
subplot(211)

# Plota a curva
# Primeira curva: ent1, sai1, 'bo:'
# Segunda curva: ent1, sai2, 'g^-'
plot(ent1, sai1, 'bo:', ent1, sai2, 'g^-')

# Cria uma legenda
legend(['Cossenos', 'Senos'])

# Seleciona a parte inferior
subplot(212)

# Desenha barras
# Eixo X: arange(len(dif)) + .5
# Eixo Y: dif
# Largura das barras: .5
# Cor: #ccbbaa
bar(arange(len(dif)) + .5, dif, .5, color='#ccbbaa')
```

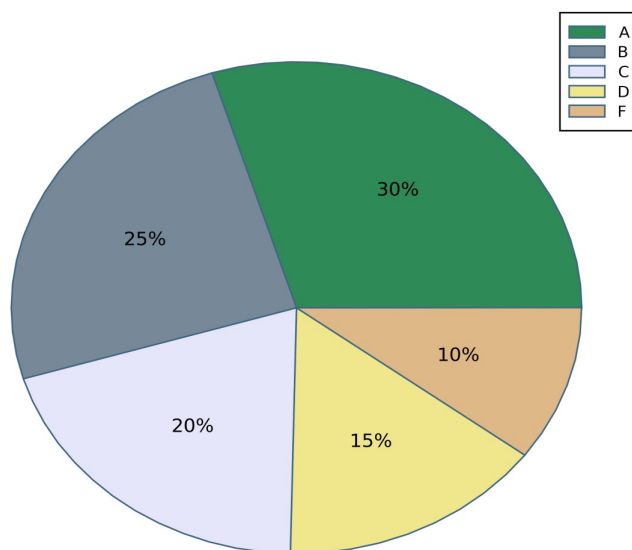
```
# Salva a figura
savefig('graf.png')
```

Saída:

O pacote tem funções para gerar gráficos de barra, linha, dispersão, pizza e polar, entre outros.

Exemplo usando *matplotlib*:

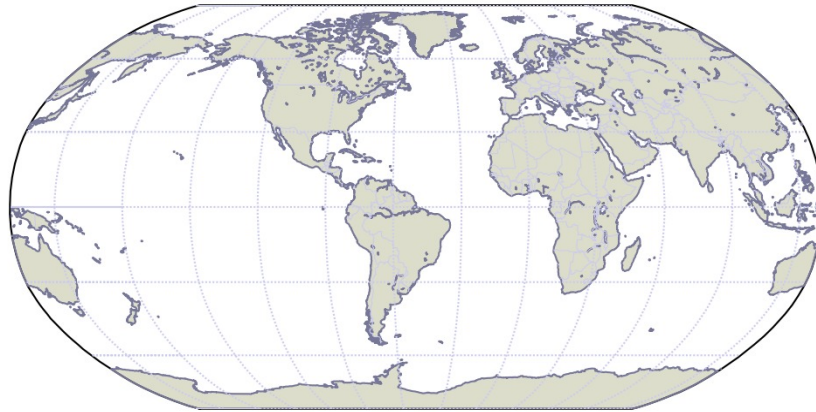
```
# -*- coding: latin1 -*-

import os

import matplotlib
from matplotlib.figure import Figure
from matplotlib.backends.backend_agg import FigureCanvasAgg

def pie(filename, labels, values):
    """
    Gera um diagrama de Pizza e salva em arquivo.
    """

    # Use a biblioteca Anti-Grain Geometry
    matplotlib.use('Agg')

    # Cores personalizadas
    colors = ['seagreen', 'lightslategray', 'lavender',
```

```
        'khaki', 'burlywood', 'cornflowerblue']

    # Altera as opções padrão
    matplotlib.rc('patch', edgecolor='#406785',
        linewidth=1, antialiased=True)
    # Altera as dimensões da imagem
    matplotlib.rc('figure', figsize=(8., 7.))

    # Inicializa a figura
    fig = Figure()
    fig.clear()
    axes = fig.add_subplot(111)

    if values:
        # Diagrama
        chart = axes.pie(values, colors=colors, autopct='%2.0f%%')

        # Legenda
        pie_legend = axes.legend(labels)
        pie_legend.pad = 0.3

        # Altera o tamanho da fonte
        for i in xrange(len(chart[0])):
            chart[-1][i].set_fontsize(12)
            pie_legend.texts[0].set_fontsize(10)

    else:
        # Mensagem de erro
        # Desliga o diagrama
        axes.set_axis_off()
        # Mostra a mensagem
        axes.text(0.5, 0.5, 'Sem dados',
            horizontalalignment='center',
            verticalalignment='center',
            fontsize=32, color='#6f7c8c')

    # Salva a figura
    canvas = FigureCanvasAgg(fig)
    canvas.print_figure(filename, dpi=600)

if __name__ == '__main__':
    # Testes
    pie('fig1.png', [], [])
    pie('fig2.png', ['A', 'B', 'C', 'D', 'E'],
        [6.7, 5.6, 4.5, 3.4, 2.3])
```

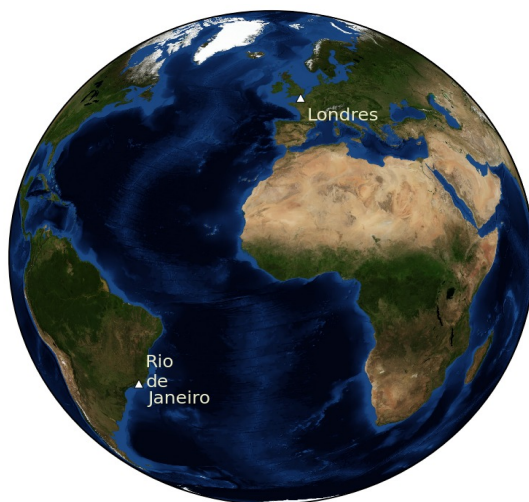
Saída:

Existem *add ons* para o Matplotlib, que expandem a biblioteca com novas funcionalidades, como é o caso do Basemap.

Exemplo com Basemap:

```
from mpl_toolkits.basemap import Basemap
from matplotlib import pyplot
from numpy import arange

# Cria um mapa usando Basemap
mapa = Basemap(projection='robin', lat_0=-20, lon_0=-50,
    resolution='l', area_thresh=1e3)

# desenha a costa dos continentes
mapa.drawcoastlines(color='#777799')
# Desenha as fronteiras
mapa.drawcountries(color='#ccccee')
# Pinta os continentes
mapa.fillcontinents(color='#ddddcc')
# Desenha os meridianos
mapa.drawmeridians(arange(0, 360, 30), color='#ccccee')
# Desenha os paralelos
mapa.drawparallels(arange(-180, 180, 30), color='#ccccee')
# Desenha os limites do mapa
mapa.drawmapboundary()

# Salva a imagem
pyplot.savefig('mapa1.png', dpi=150)
```

Saída:

Outro exemplo:

```
from mpl_toolkits.basemap import Basemap
from matplotlib import pyplot

mapa = Basemap(projection='ortho', lat_0=10, lon_0=-10,
    resolution='l', area_thresh=1e3)
# Preenche o mapa com relevo
mapa.bluemarble()
mapa.drawmapboundary()

lxy = (('Rio\nde\nJaneiro', -43.11, -22.54),
    ('Londres', 0.07, 50.30))

# Transposta
lxy = zip(*lxy)
# Converte as coordenadas
x, y = mapa(lxy[1], lxy[2])
lxy = lxy[0], x, y
# Marca no mapa
mapa.plot(x, y, 'w^')

# Escreve os nomes
for l, x, y in zip(*lxy):
    pyplot.text(x+2e5, y-6e5, l,
        color='#eeeecc')

pyplot.savefig('mapa2.png', dpi=150)
```

Saída:

Para processamento de informações georeferenciadas mais sofisticados, existe o projeto MapServer[22], que é um servidor de aplicação voltado para GIS (Geographic Information System) que suporta diversas linguagens, inclusive Python.

Além de módulos de terceiros, também é possível usar o BrOffice.org[23] para gerar gráficos com o Python, através da API chamada Python-UNO Bridge[24].

22 Site oficial em http://mapserver.org/.
23 Disponível em: http://www.broffice.org/.
24 Mais informações em: http://udk.openoffice.org/python/python-bridge.html.

Processamento de imagem

Python Imaging Library[25] (PIL) é uma biblioteca de processamento de imagens *raster* para Python.

PIL possui módulos que implementam:

- Ferramentas para cortar, redimensionar e mesclar imagens.
- Algoritmos de conversão.
- Filtros, tais como suavizar, borrar e detectar bordas.
- Ajustes, incluindo brilho e contraste.
- Operações com paletas de cores.
- Desenhos simples em 2D.
- Rotinas para tratamento de imagens: equalização, auto-contraste, deformar, inverter e outras.

Exemplo de tratamento de imagem:

```
# -*- coding: latin-1 -*-
"""
Cria miniaturas suavizadas para cada
JPEG na pasta corrente
"""

import glob

# Módulo principal do PIL
import Image

# Módulo de filtros
import ImageFilter

# Para cada arquivo JPEG
for fn in glob.glob("*.jpg"):

    # Retorna o nome do arquivo sem extensão
    f = glob.os.path.splitext(fn)[0]

    print 'Processando:', fn
    imagem = Image.open(fn)

    # Cria thumbnail (miniatura) da imagem
    # de tamanho 256x256 usando antialiasing
    imagem.thumbnail((256, 256), Image.ANTIALIAS)

    # Filtro suaviza a imagem
```

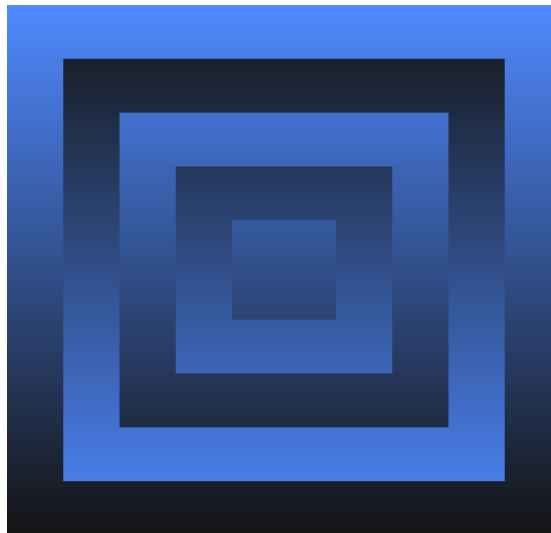
25 Documentação, fontes e binários disponíveis em: http://www.pythonware.com/products/pil/.

```
    imagem = imagem.filter(ImageFilter.SMOOTH)

    # Salva como arquivo PNG
    imagem.save(f + '.png', 'PNG')
```

Exemplo de desenho:

```
# -*- coding: latin-1 -*-
"""
Cria uma imagem com vários gradientes de cores
"""

import Image

# Módulo de desenho
import ImageDraw

# Largura e altura
l, a = 512, 512

# Cria uma imagem nova com fundo branco
imagem = Image.new('RGBA', (l, a), 'white')

# O objeto desenho age sobre o objeto imagem
desenho = ImageDraw.Draw(imagem)

# Calcula a largura da faixa de cor
faixa = l / 256

# Desenha um gradiente de cor
for i in xrange(0, l):

    # Calcula a cor da linha
    rgb = (0.25 * i / faixa, 0.5 * i / faixa, i / faixa)
    cor = '#%02x%02x%02x' % rgb

    # Desenha uma linha colorida
    # Primeiro argumento é uma tupla com
    # as coordenadas de inicio e fim da linha
    desenho.line((0, i, l, i), fill=cor)

# Copia e cola recortes invertidos do gradiente
for i in xrange(l, l / 2, -l / 10):

    # Tamanho do recorte
    area = (l - i, a - i, i, i)

    # Copia e inverte
    flip = Image.FLIP_TOP_BOTTOM
    recorte = imagem.crop(area).transpose(flip)
```

```
    # Cola de volta na imagem original
    imagem.paste(recorte, area)

# Salva como arquivo PNG
imagem.save('desenho.png', 'PNG')
```

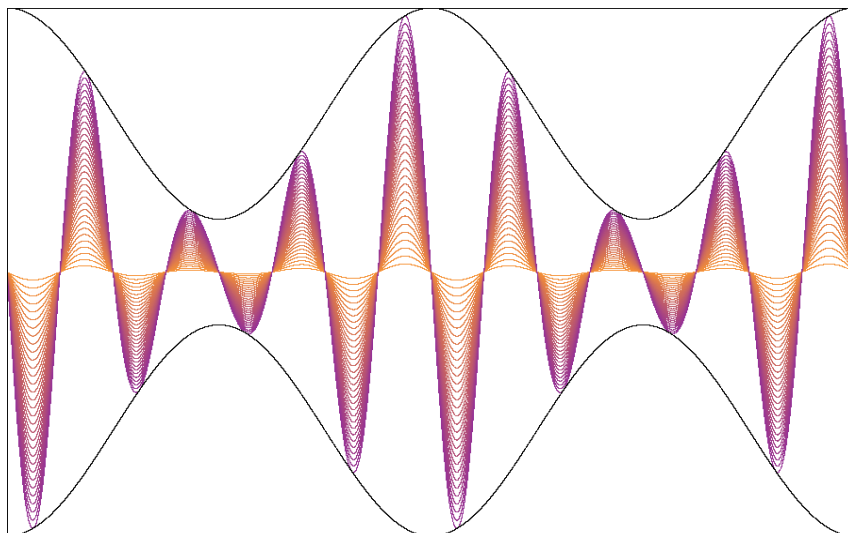
Arquivo de saída “desenho.png”:

É possível usar o NumPy para calcular os dados da imagem e usar o PIL para gerar a imagem real.

Exemplo com modulação de amplitude de onda :

```
# -*- coding: latin1 -*-
"""
Criando uma imagem usando NumPy
"""

import numpy
import Image

def coords(xy, tam):
    """
    coords(xy, tam) => x, y
    Transforma as coordenadas normalizadas
    para o centro da imagem de tamanho "tam"
    """
    X, Y = tam

    x = int((1. + xy[0]) * (X - 1.) / 2.)
```

```
    y = int((1. + xy[1]) * (Y - 1.) / 2.)
    return x, y

if __name__ == '__main__':

    # Dimensões
    tam = 900, 600

    # Cria um arranjo apenas com zeros
    # com as dimensões transpostas
    # "tam[::-1]" é o reverso de "tam" e
    # "(3,)" é uma tupla para representar "(R, G, B)"
    imag = numpy.zeros(tam[::-1] + (3,), numpy.uint8)

    # Preenche de branco
    imag.fill(255)

    # Dados do eixo X
    xs = numpy.arange(-1., 1., 0.00005)

    # Onda moduladora
    # Valor médio, amplitude e freqüência
    vmed = 0.6
    amp = 0.4
    fm = 2.
    mod = vmed + amp * numpy.cos(fm * numpy.pi * xs)

    # Freqüência da portadora
    fc = 8.
    # Número de curvas internas
    ci = 32.
    # Contador
    i = 0

    # Gera um conjunto de curvas
    for delta_y in numpy.arange(1. / ci, 1. + 1. / ci,
        1. / ci):

        # Dados do eixo Y
        ys = mod * delta_y * numpy.sin(fc * numpy.pi * xs)

        # Pares x, y
        xys = zip(xs, ys)

        # Desenha a portadora e as curvas internas
        # Para cada ponto na lista
        for xy in xys:

            # Coordenadas invertidas
            x, y = coords(xy, tam)[::-1]

            # Aplica cor a xy
```

```
            imag[x, y] = (250 - 100 * delta_y,
                150 - 100 * delta_y,
                50 + 100 * delta_y)
            i += 1

    for x, y in zip(xs, mod):
        # Desenha as envoltórias
        imag[coords((x, y), tam)[::-1]] = (0, 0, 0)
        imag[coords((x, -y), tam)[::-1]] = (0, 0, 0)

        # Bordas superior e inferior
        imag[coords((x, 1.), tam)[::-1]] = (0, 0, 0)
        imag[coords((x, -1.), tam)[::-1]] = (0, 0, 0)
        i += 4

    for y in xs:

        # Bordas laterais
        imag[coords((1., y), tam)[::-1]] = (0, 0, 0)
        imag[coords((-1., y), tam)[::-1]] = (0, 0, 0)
        i += 2

    print i, 'pontos calculados'

    # Cria a imagem a partir do arranjo
    imagem = Image.fromarray(imag, 'RGB')
    imagem.save('curvas.png', 'PNG')
```

Arquivo de saída "curvas.png":

Observações:

- A biblioteca trabalha com o conceito de bandas, que são camadas que compõem a imagem. Cada imagem pode ter várias bandas, mas todas devem ter as mesmas

dimensões e profundidade.

- A origem do sistema de coordenadas é no canto superior esquerdo.

Gráficos 3D

VPython[26] é um pacote que permite criar e animar modelos simples em três dimensões. Seu objetivo é facilitar a criação rápida de simulações e protótipos que não requerem soluções complexas.

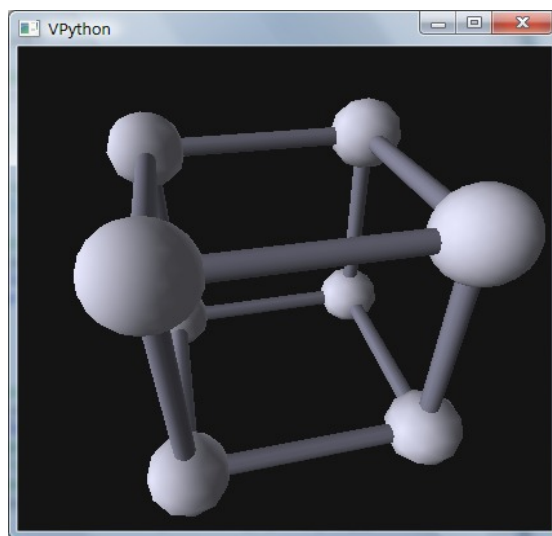
O VPython provê iluminação, controle de câmera e tratamento de eventos de mouse (rotação e *zoom*) automaticamente. Os objetos podem ser criados interativamente no interpretador que a vista tridimensional é atualizada de acordo.

Exemplo:

```
# -*- coding: latin-1 -*-
"""
Hexaedro
"""

# VPython
import visual

# Coordenadas para os vértices e arestas
coords = (-3, 3)

# Cor do vértice
cor1 = (0.9, 0.9, 1.0)

# Cor da aresta
cor2 = (0.5, 0.5, 0.6)

# Desenha esferas nos vértices
for x in coords:
   for y in coords:
      for z in coords:
         # pos é a posição do centro da esfera
         visual.sphere(pos=(x, y, z), color=cor1)

# Desenha os cilindros das arestas
for x in coords:
   for z in coords:
      # pos é a posição do centro da base do cilindro
      # radius é o raio da base do cilindro
      # axis é o eixo do cilindro
      visual.cylinder(pos=(x, 3, z), color=cor2,
         radius=0.25, axis=(0, -6, 0))
```

26 Documentação, fontes e binários para instalação em: http://www.vpython.org/.

```
    for y in coords:
        visual.cylinder(pos=(x, y, 3), color=cor2,
            radius=0.25, axis=(0, 0, -6))

for y in coords:
    for z in coords:
        visual.cylinder(pos=(3, y, z), color=cor2,
            radius=0.25, axis=(-6, 0, 0))
```

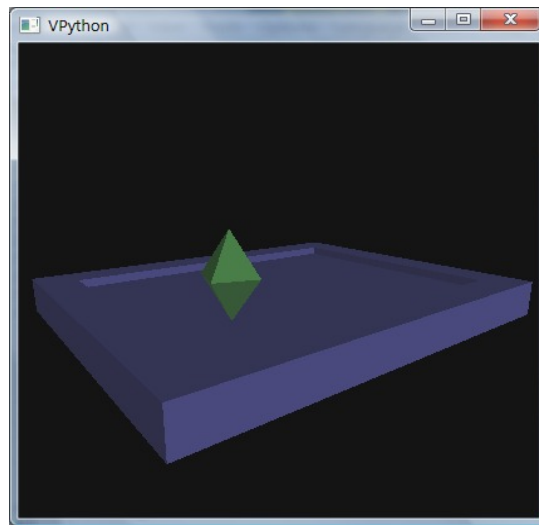
Janela 3D:

Os objetos 3D do VPython podem ser agrupados em quadros (*frames*), que podem ser movidos e rotacionados.

É possível animar os objetos 3D usando laços. Para controlar a velocidade da animação, o VPython provê a função *rate()*, que pausa animação pelo inverso do argumento em segundos.

Exemplo de quadro e animação:

```
# -*- coding: latin-1 -*-
"""
Octaedro animado
"""

from visual import *

# Cores
azul = (0.25, 0.25, 0.50)
```

```
verde = (0.25, 0.50, 0.25)

# Eixo de rotação
eixo = (0, 1, 0)

# Cria um frame alinhado com o eixo de rotação
fr = frame(axis=eixo)

# O fundo da caixa
box(pos=(0, -0.5, 0), color=azul,
   size=(10.0, 0.5, 8.0))

# O bordas da caixa
box(pos=(0, -0.5, 4.0), color=azul,
   size=(11.0, 1.0, 1.0))
box(pos=(0, -0.5, -4.0), color=azul,
   size=(11.0, 1.0, 1.0))
box(pos=(5.0, -0.5, 0), color=azul,
   size=(1.0, 1.0, 8.0))
box(pos=(-5.0, -0.5, 0), color=azul,
   size=(1.0, 1.0, 8.0))

# O pião
py1 = pyramid(frame=fr, pos=(1, 0, 0), color=verde,
   axis=(1, 0, 0))
py2 = pyramid(frame=fr, pos=(1, 0, 0), color=verde,
   axis=(-1, 0, 0))

# O pião anda no plano y = 0
delta_x = 0.01
delta_z = 0.01

print fr.axis

while True:

    # Inverte o sentido em x
    if abs(fr.x) > 4.2:
        delta_x = -delta_x

    # Inverte o sentido em z
    if abs(fr.z) > 3.1:
        delta_z = -delta_z

    fr.x += delta_x
    fr.z += delta_z

    # Rotaciona em Pi / 100 no eixo
    fr.rotate(angle=pi / 100, axis=eixo)

    # Espere 1 / 100 segundos
    rate(250)
```

Janela 3D:

O VPython tem várias limitações. Ele não provê formas de criar e/ou manipular materiais ou texturas sofisticadas, nem formas avançadas de iluminação ou detecção de colisões. Para modelagens mais sofisticadas, existem outras soluções, como o Python Ogre[27] e o Blender, que é um aplicativo de modelagem que usa Python como linguagem *script*.

27 Disponível em: http://python-ogre.org/.

Persistência

Persistência pode ser definida como a manutenção do estado de uma estrutura de dados entre execuções de uma aplicação. A persistência libera o desenvolvedor de escrever código explicitamente para armazenar e recuperar estruturas de dados em arquivos e ajuda a manter o foco na lógica da aplicação.

Serialização

A forma mais simples e direta de persistência é chamada de serialização[28] e consiste em gravar em disco uma imagem (*dump*) do objeto, que pode recarregada (*load*) posteriormente. No Python, a serialização é implementada de várias formas, sendo que a mais comum é através do módulo chamado *pickle*.

Exemplo de serialização:

- O programa tenta recuperar o dicionário *setup* usando o objeto do arquivo "setup.pkl".
- Se conseguir, imprime o dicionário.
- Se não conseguir, cria um *setup default* e salva em "setup.pkl".

```
import pickle

try:
    setup = pickle.load(file('setup.pkl'))
    print setup

except:
    setup = {'timeout': 10,
        'server': '10.0.0.1',
        'port': 80
        }
    pickle.dump(setup, file('setup.pkl', 'w'))
```

Na primeira execução, ele cria o arquivo. Nas posteriores, a saída é:

```
{'port': 80, 'timeout': 10, 'server': '10.0.0.1'}
```

Entre os módulos da biblioteca padrão estão disponíveis outros módulos persistência, tais como:

- *cPickle*: versão mais eficiente de *pickle*, porém não pode ter subclasses.
- *shelve*: fornece uma classe de objetos persistentes similares ao dicionário.

28 Em inglês, *serialization* ou *marshalling*.

Existem *frameworks* em Python de terceiros que oferecem formas de persistência com recursos mais avançados, como o ZODB.

Todas essas formas de persistência armazenam dados em formas binárias, que não são diretamente legíveis por seres humanos.

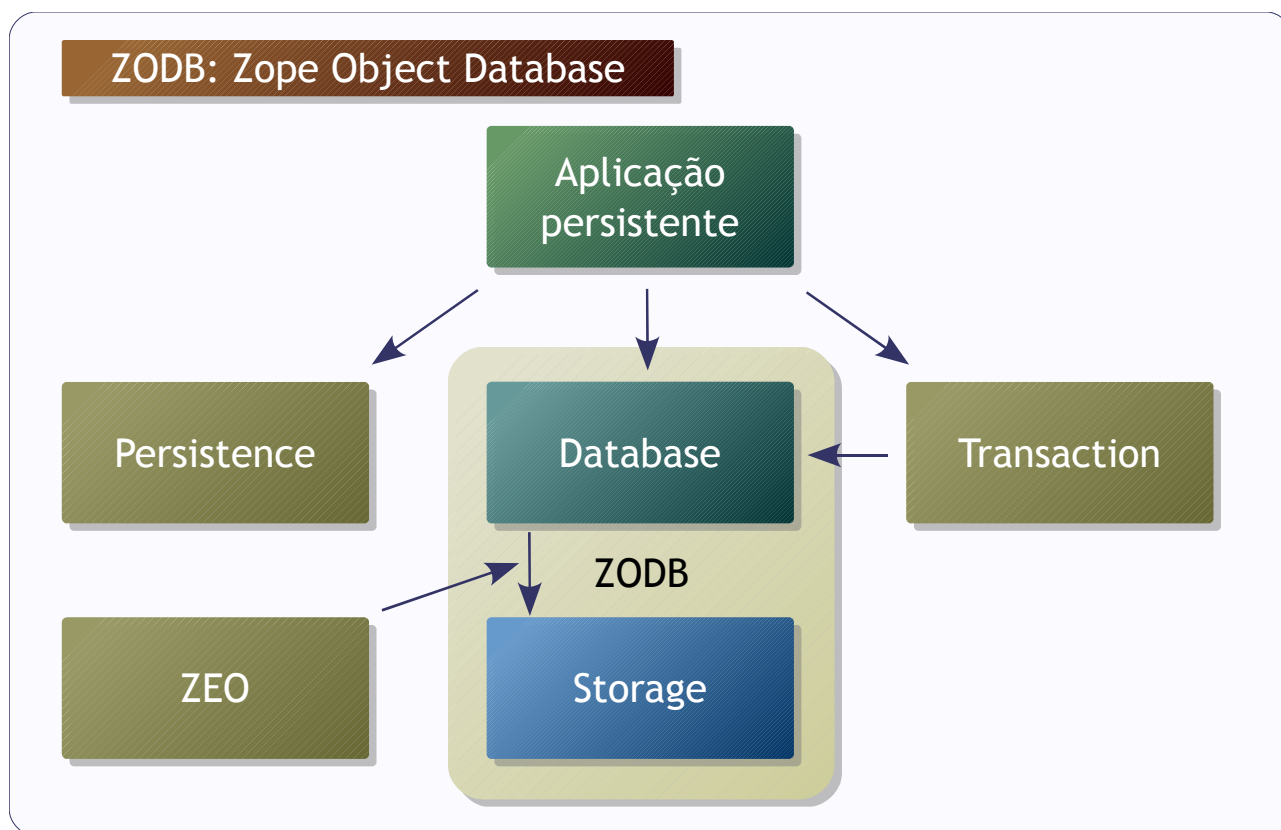
Para armazenar dados de forma de texto, existem módulos para Python para ler e gravar estruturas de dados em formatos:

- JSON[29] (*JavaScript Object Notation*).
- YAML[30] (*YAML Ain't a Markup Language*).
- XML[31] (*Extensible Markup Language*).

29 Página do formato em: http://www.json.org/.
30 Página do formato em: http://yaml.org/.
31 Página do formato em: http://www.w3.org/XML/.

ZODB

Zope Object Database (ZODB) é um banco de dados orientado a objeto que oferece uma forma de persistência quase transparente para aplicações escritas em Python e foi projetado para ter pouco impacto no código da aplicação.

ZODB suporta transações, controle de versão de objetos e pode ser conectado a outros *backends* através do *Zope Enterprise Objects* (ZEO), permitindo inclusive a criação de aplicações distribuídas em diversas máquinas conectadas por rede.

O ZODB é um componente integrante do Zope[32], que é um servidor de aplicações desenvolvido em Python, muito usado em *Content Management Systems* (CMS).

Componentes do ZODB:

- *Database*: permite que a aplicação abra conexões (interfaces para acesso aos objetos).
- *Transaction*: interface que permite tornar as alterações permanentes.
- *Persistence* : fornece a classe base Persistent.
- *Storage*: gerencia a representação persistente em disco.
- *ZEO*: compartilhamento de objeto entre diferentes processos e máquinas.

---

32 Documentação e pacotes de instalação do Zope e produtos ligados a ele em http://www.zope.org/.

Exemplo de uso do ZODB:

```
# -*- coding: latin1 -*-

from ZODB import FileStorage, DB
import transaction

# Definindo o armazenamento do banco
storage = FileStorage.FileStorage('people.fs')
db = DB(storage)

# Conectando
conn = db.open()

# Referência para a raíz da árvore
root = conn.root()

# Um registro persistente
root['singer'] = 'Kate Bush'

# Efetuando a alteração
transaction.commit()
print root['singer'] # Kate Bush

# Mudando um atributo
root['singer'] = 'Tori Amos'
print root['singer'] # Tori Amos

# Abortando...
transaction.abort()
print root['singer'] # Kate Bush
```

O ZODB tem algumas limitações que devem ser levadas em conta durante o projeto da aplicação:

- Os objetos precisam ser "serializáveis" para serem armazenados.
- Objetos mutáveis requerem cuidados especiais.

Objetos "serializáveis" são aqueles objetos que podem ser convertidos e recuperados pelo *Pickle*. Entres os objetos que não podem ser processados pelo *Pickle*, estão os objetos implementados em módulos escritos em C, por exemplo.

YAML

YAML é um formato de serialização de dados para texto que representa os dados como combinações de listas, dicionários e valores escalares. Tem como principal característica ser legível por humanos.

O projeto do YAML foi muito influenciado pela sintaxe do Python e outras linguagens dinâmicas. Entre outras estruturas, a especificação[33] do YAML define que:

- Os blocos são marcados por endentação.
- Listas são delimitadas por colchetes ou indicadas por traço.
- Chaves de dicionário são seguidas de dois pontos.

Listas podem ser representadas assim:

```
- Azul
- Branco
- Vermelho
```

Ou:

```
[azul, branco, vermelho]
```

Dicionários são representados como:

```
cor: Branco
nome: Bandit
raca: Labrador
```

PyYAML[34] é uma biblioteca de rotinas para gerar e processar YAML no Python.

Exemplo de conversão para YAML:

```
import yaml

progs = {'Inglaterra':
         {'Yes': ['Close To The Edge', 'Fragile'],
         'Genesis': ['Foxtrot', 'The Nursery Crime'],
```

33 Disponível em: http://yaml.org/spec/1.2/.
34 Documentação e fontes em: http://pyyaml.org/wiki/PyYAML.

```
        'King Crimson': ['Red', 'Discipline']},
    'Alemanha':
        {'Kraftwerk': ['Radioactivity', 'Trans Europe Express']}
    }

print yaml.dump(progs)
```

Saída:

```
Alemanha:
  Kraftwerk: [Radioactivity, Trans Europe Express]
Inglaterra:
  Genesis: [Foxtrot, The Nursery Crime]
  King Crimson: [Red, Discipline]
  'Yes': [Close To The Edge, Fragile]
```

Exemplo de leitura de YAML. Arquivo de entrada "prefs.yaml":

```
- musica: rock
- cachorro:
   cor: Branco
   nome: Bandit
   raca: Labrador
- outros:
   instrumento: baixo
   linguagem: [python, ruby]
   comida: carne
```

Código em Python:

```
import pprint
import yaml

# yaml.load() pode receber um arquivo aberto
# como argumento
yml = yaml.load(file('prefs.yaml'))

# pprint.pprint() mostra a estrutura de dados
# de uma forma mais organizada do que
# o print convencional
pprint.pprint(yml)
```

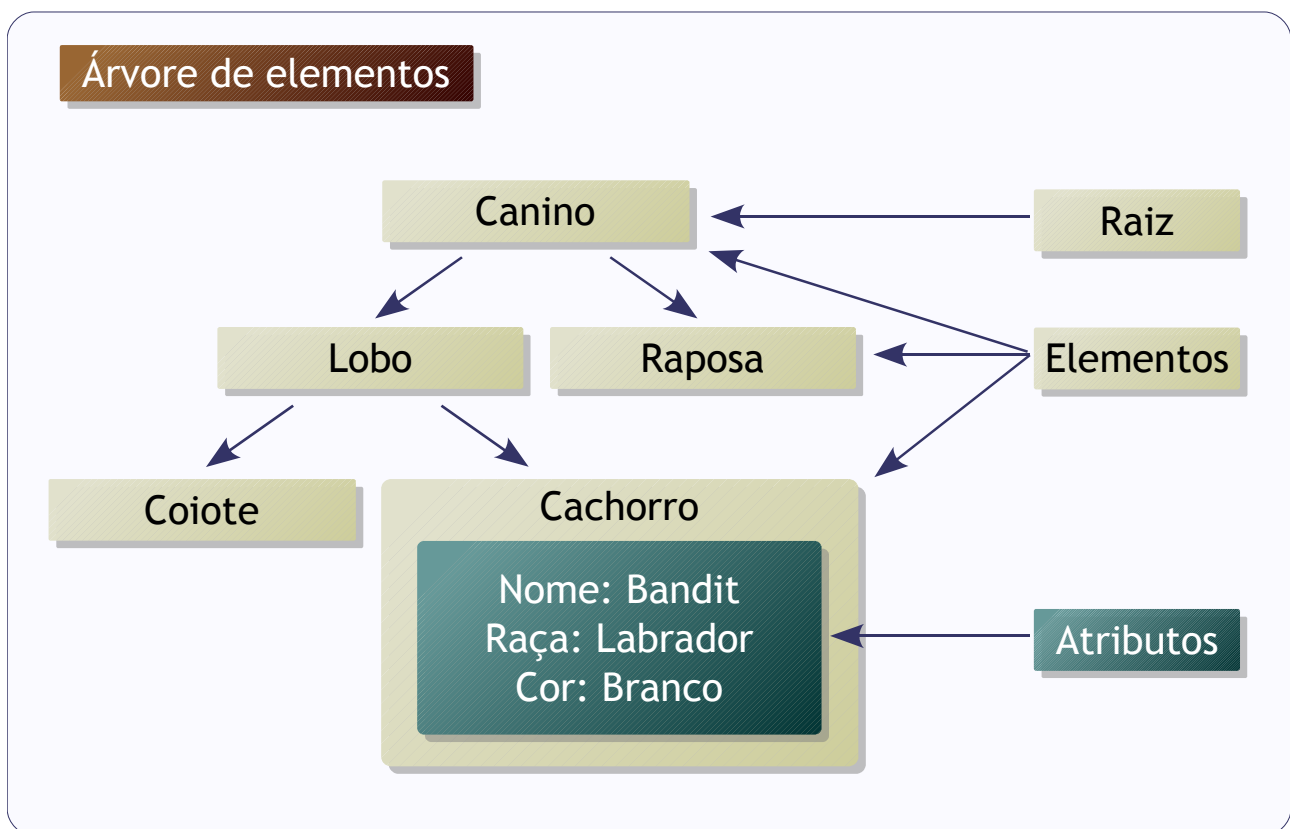
Saída:

```
[{'musica': 'rock'},
 {'cachorro': {'cor': 'Branco', 'nome': 'Bandit', 'raca': 'Labrador'}},
 {'outros': {'comida': 'carne',
             'instrumento': 'baixo',
             'linguagem': ['python', 'ruby']}}]
```

YAML é muito prático para ser usado em arquivos de configuração e outros casos onde os dados podem ser manipulados diretamente por pessoas.

XML

XML (*eXtensible Markup Language*) é uma recomendação, desenvolvida pelo *World Wide Web Consortium*[35] (W3C), para uma representação de dados em que o metadado é armazenado junto com os dados através de marcadores (*tags*).

Em termos estruturais, um arquivo XML representa uma hierarquia formada de elementos, que podem ter ou não atributos ou sub elementos.

Características principais:

- É legível por software.
- Pode ser integrada com outras linguagens.
- O conteúdo e a formatação são entidades distintas.
- Marcadores podem ser criados sem limitação.
- Permite a criação de arquivos para validação de estrutura.

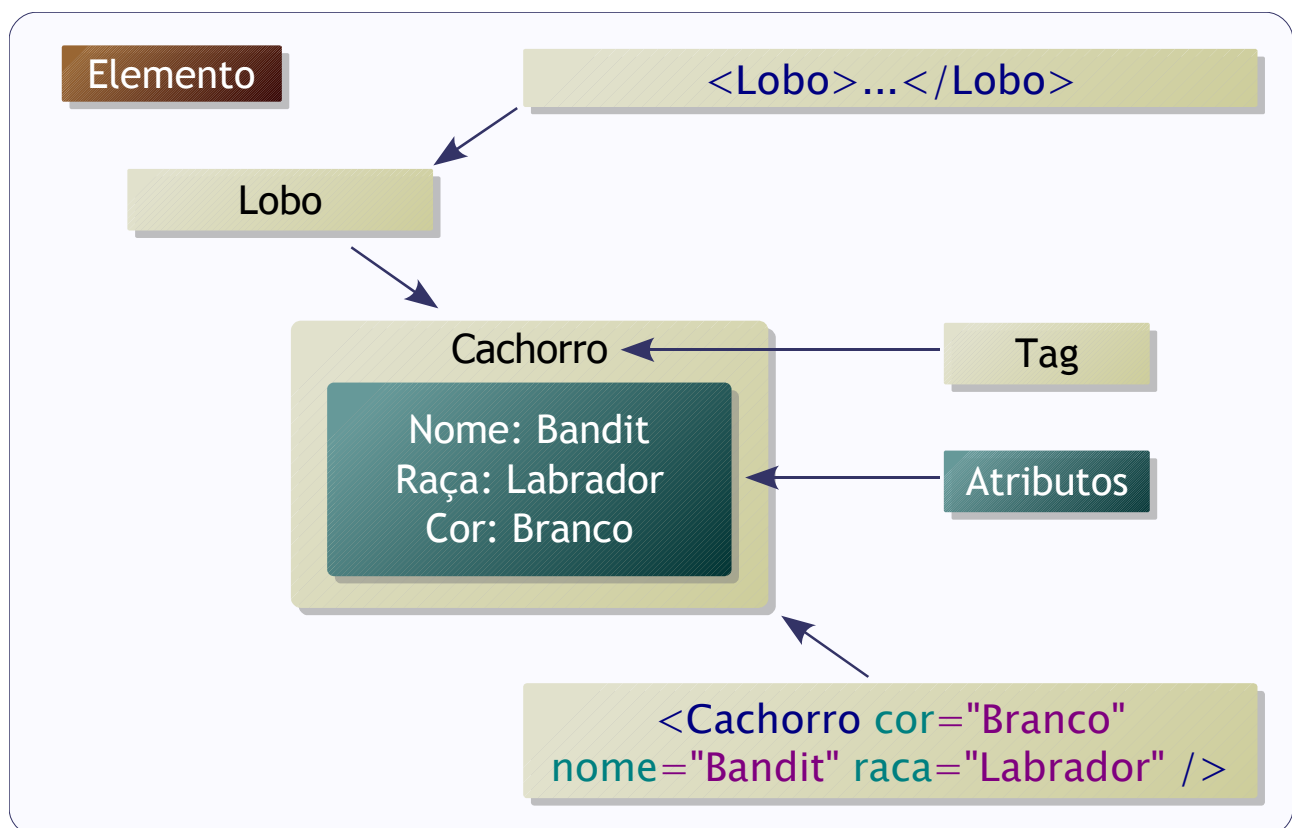
No exemplo, o elemento "Cachorro" possui três atributos: nome, raça e cor. O elemento Lobo tem dois sub elementos ("Cachorro" e "Coiote") e não possui atributos.

35 Página oficial em: http://www.w3.org/.

Em XML, o cachorro é representado por:

```
<Cachorro cor="Branco" nome="Bandit" raca="Labrador" />
```

E o lobo por:

```
<Lobo>   </Lobo>
```

Existem vários módulos de suporte ao XML disponíveis para Python, inclusive na biblioteca que acompanha o interpretador.

Entre os mais usados, destacam-se:

- DOM.
- SAX.
- *ElementTree*.

Document Object Model (DOM) é um modelo de objeto para representação de XML, independente de plataforma e linguagem. O DOM foi projetado para permitir navegação não linear e modificações arbitrárias. Por isso, o DOM exige que o documento XML (ou pelo menos parte dele) esteja carregado na memória.

Exemplo:

```
# -*- coding: latin1 -*-

# importa a implementação minidom
import xml.dom.minidom

# Cria o documento
doc = xml.dom.minidom.Document()

# Para ler um documento que já existe
# doc = xml.dom.minidom.parse('caninos.xml')

# Cria os elementos
root = doc.createElement('Canino')
lobo = doc.createElement('Lobo')
raposa = doc.createElement('Raposa')
coiote = doc.createElement('Coiote')
cachorro = doc.createElement('Cachorro')

# Cria os atributos
cachorro.setAttribute('nome', 'Bandit')
cachorro.setAttribute('raca', 'Labrador')
cachorro.setAttribute('cor', 'Branco')

# Cria a estrutura
doc.appendChild(root)
root.appendChild(lobo)
root.appendChild(raposa)
lobo.appendChild(coiote)
lobo.appendChild(cachorro)

# Para acrescentar texto ao elemento
# tex = doc.createTextNode('Melhor amigo do homem...')
# cachorro.appendChild(tex)

# Mostra o XML formatado
print doc.toprettyxml()
```

Simple API for XML (SAX) é uma API de análise sintática serial para XML. SAX permite apenas a leitura serial do documento XML. SAX é mais eficiente que o DOM, porém tem menos recursos.

Exemplo:

```
# -*- coding: latin1 -*-
```

```
import xml.sax


# A classe processa o árvore XML
class Handler(xml.sax.handler.ContentHandler):

    def __init__(self):

        xml.sax.handler.ContentHandler.__init__(self)
        self.prefixo = ''

    # É chamado quando uma novo tag é encontrada
    def startElement(self, tag, attr):

        self.prefixo += '  '
        print self.prefixo + 'Elemento:', tag
        for item in attr.items():
            print self.prefixo + '- %s: %s' % item

    # É chamado quando texto é encontrado
    def characters(self, txt):

        if txt.strip():
            print self.prefixo + 'txt:', txt

    # É chamado quando o fim de uma tag é encontrada
    def endElement(self, name):

        self.prefixo = self.prefixo[:-2]


parser = xml.sax.make_parser()
parser.setContentHandler(Handler())
parser.parse('caninos.xml')
```

ElementTree é o mais “pythônico” dos três, representando uma estrutura XML como uma árvore de elementos, que são tratados de forma semelhante às listas, e nos quais os atributos são chaves, similar aos dicionários.

Exemplo de geração de XML com *ElementTree*:

```
from xml.etree.ElementTree import Element, ElementTree

root = Element('Canino')
lobo = Element('Lobo')
raposa = Element('Raposa')
coiote = Element('Coiote')
cachorro = Element('Cachorro', nome='Bandit',
    raca='Labrador', cor='Branco')
```

```
root.append(lobo)
root.append(raposa)
lobo.append(coiote)
lobo.append(cachorro)

ElementTree(root).write('caninos.xml')
```

Arquivo XML de saída:

```
<Canino>
   <Lobo>
      <Coiote />
      <Cachorro cor="Branco" nome="Bandit" raca="Labrador" />
   </Lobo>
   <Raposa />
</Canino>
```

Exemplo de leitura do arquivo XML:

```
from xml.etree.ElementTree import ElementTree

tree = ElementTree(file='caninos.xml')
root = tree.getroot()

# Lista os elementos abaixo do root
print root.getchildren()

# Encontra o lobo
lobo = root.find('Lobo')

# Encontra o cachorro
cachorro = lobo.find('Cachorro')
print cachorro.tag, cachorro.attrib

# Remove a raposa
root.remove(root.find('Raposa'))
print root.getchildren()
```

Saída:

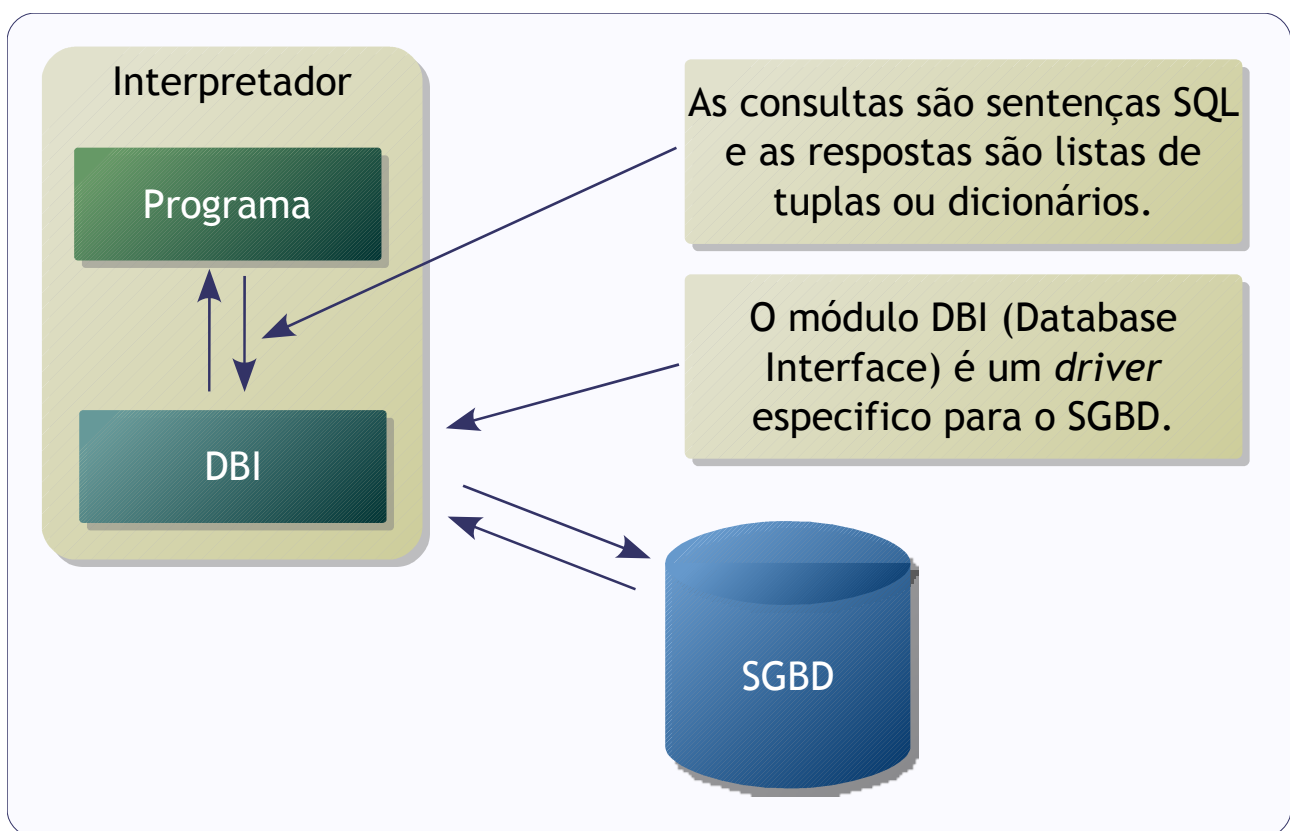
```
[<Element Lobo at ab3a58>, <Element Raposa at ab3b70>]
Cachorro {'cor': 'Branco', 'raca': 'Labrador', 'nome': 'Bandit'}
[<Element Lobo at ab3a58>]
```

O XML é muito útil por facilitar a interoperabilidade entre sistemas, mesmo que estes sejam desenvolvidos em tecnologias diferentes.

Banco de dados

Sistemas Gerenciadores de Banco de Dados (SGBDs) são reconhecidos por prover uma forma de acesso consistente e confiável para informações. No Python, a integração com SGBDs geralmente é feita através de um módulo DBI.

DBI

Database Interface (DBI) é uma especificação que descreve como deve ser o comportamento de um módulo de acesso a sistemas de banco de dados.

A DBI define que o módulo deve ter uma função *connect()*, retorna objetos de conexão. A partir do do objeto conexão, é possível obter um objeto cursor, que permite a execução de sentenças SQL.

Exemplo de acesso através de DBI (com MySQL[36]):

```
# -*- coding: utf-8 -*-
```

36 Binários, fontes e documentação podem ser encontrados em: http://sourceforge.net/projects/mysql-python.

```
import MySQLdb

# Cria uma conexão
con = MySQLdb.connect(db='test', user='root', passwd='')

# Cria um cursor
cur = con.cursor()

# Executa um comando SQL
cur.execute('show databases')

# Recupera o resultado
recordset = cur.fetchall()

# Mostra o resultado
for record in recordset:
    print record

# Fecha a conexão
con.close()
```

Saída:

```
('information_schema',)
('mysql',)
('test',)
```

O resultado é uma lista de tuplas com as *databases* disponíveis no servidor.

SQLite

A partir da versão 2.5, o Python passou a incorporar em sua distribuição um módulo DBI para acessar o SQLite[37].

SQLite é uma biblioteca *Open Source* escrita em linguagem C, que implementa um interpretador SQL, que provê funcionalidades de banco de dados, usando arquivos, sem a necessidade de um processo servidor separado ou de configuração manual.

Exemplo:

```
# -*- coding: utf-8 -*-
```

37 Documentação, fontes e binários podem ser encontrados em: http://www.sqlite.org/.

```
import sqlite3

# Cria uma conexão e um cursor
con = sqlite3.connect('emails.db')
cur = con.cursor()

# Cria uma tabela
sql = 'create table emails '\
   '(id integer primary key, '\
   'nome varchar(100), '\
   'email varchar(100))'
cur.execute(sql)

# sentença SQL para inserir registros
sql = 'insert into emails values (null, ?, ?)'

# Dados
recset = [('jane doe', 'jane@nowhere.org'),
   ('rock', 'rock@hardplace.com')]

# Insere os registros
for rec in recset:
   cur.execute(sql, rec)

# Confirma a transação
con.commit()

# Seleciona todos os registros
cur.execute('select * from emails')

# Recupera os resultados
recset = cur.fetchall()

# Mostra
for rec in recset:
   print '%d: %s(%s)' % rec

# Fecha a conexão
con.close()
```

A vantagem mais significativa de usar o SQLite é a praticidade, principalmente no uso em aplicativos locais para *desktops*, aonde usar um SGBD convencional seria desnecessário e complicado de manter.

PostgreSQL

Para sistemas que demandam recursos mais sofisticados do SGBD, o PostgreSQL[38] é a

38 Site oficial em http://www.postgresql.org/ e site da comunidade brasileira em http://www.postgresql.org.br/.

solução Open Source mais completa disponível.

Entre os recursos oferecidos pelo PostgreSQL, destacam-se:

- Suporte a consultas complexas.
- Transações.
- Controle de concorrência multi-versão.
- Tipos de objetos definidos pelo usuário.
- Herança.
- *Views*.
- *Stored Procedures*.
- *Triggers*.
- *Full text search*.

Existem vários módulos que provêm acesso ao PostgreSQL para o Python, como o PygreSQL[39] e o Psycopg[40].

O PygreSQL oferece duas interfaces distintas para acesso a servidores PostgreSQL:

- pgdb: módulo compatível com DBI.
- pg: módulo mais antigo, incompatível com DBI.

Exemplo com pgdb:

```
# -*- coding: latin1 -*-

import pgdb

# Para bancos de dados locais (via Unix Domain Sockets)
#con = pgdb.connect(database='music')

# Via TCP/IP
con = pgdb.connect(host='tao', database='music', user='pg', password='#@!$%&')
cur = con.cursor()

# Cria uma tabela
sql = 'create table tracks '\
  '(id serial primary key, '\
  'track varchar(100), '\
  'band varchar(100))'
cur.execute(sql)

# A interpolação usa uma notação semelhante a do Python
sql = 'insert into tracks values (default, %s, %s)'
```

39 Site oficial: http://www.pygresql.org/.
40 Fontes e documentação em http://initd.org/.

```
# Dados
recset = [('Kashmir', 'Led Zeppelin'),
  ('Starless', 'King Crimson')]

# Insere os registros
for rec in recset:
  cur.execute(sql, rec)

con.commit()

# Recupera os registros
cur.execute('select * from tracks')

# Recupera os resultados
recset = cur.fetchall()
# Mostra
for rec in recset:
  print rec

con.close()
```

Saída:

```
[1, 'Kashmir', 'Led Zeppelin']
[2, 'Starless', 'King Crimson']
```

Exemplo com pg:

```
import pg
# Para bancos de dados locais (via Unix Domain Sockets)
#con = pg.connect('music')

# Via TCP/IP
con = pg.connect(host='tao', dbname='music', user='pg', passwd='#@!$%&')

# Realiza uma consulta no banco
qry = con.query('select * from tracks')

# Pega a lista de campos
flds = qry.listfields()

# Mostra os resultados
for rec in qry.dictresult():
  for fld in flds:
     print '%s: %s' % (fld, rec[fld])
  print

con.close()
```

Saída:

```
id: 1
track: Kashmir
band: Led Zeppelin

id: 2
track: Starless
band: King Crimson
```

Exemplo usando o módulo psycopg:

```
import psycopg2

# Para bancos de dados locais (via Unix Domain Sockets)
#con = psycopg2.connect(database='music')

# Via TCP/IP
con = psycopg2.connect(host='tao', database='music',
  user='pg', password='#@!$%&')
cur = con.cursor()

sql = 'insert into tracks values (default, %s, %s)'
recset = [('Siberian Khatru', 'Yes'),
  ("Supper's Ready", 'Genesis')]
for rec in recset:
  cur.execute(sql, rec)
con.commit()

cur.execute('select * from tracks')
recset = cur.fetchall()
for rec in recset:
  print rec

con.close()
```

Saída:

```
(1, 'Kashmir', 'Led Zeppelin')
(2, 'Starless', 'King Crimson')
(3, 'Siberian Khatru', 'Yes')
(4, "Supper's Ready", 'Genesis')
```

Como o módulo segue fielmente a especificação DBI, o código é praticamente igual ao exemplo usando o módulo pg. O psycopg foi projetado com o objetivo de suportar

aplicações mais pesadas, com muitas inserções e atualizações.

Também é possível escrever funções para PostgreSQL usando Python. Para que isso seja possível, é preciso habilitar o suporte ao Python no banco, através do utilitário de linha de comando pelo administrador:

```
createlang plpythonu <banco>
```

As linguagens que podem usadas pelo PostgreSQL são chamadas *Procedural Languages* (PL) e o sufixo "u" significa *untrusted*.

Os tipos dos parâmetros e do retorno da função devem ser definidos durante a criação da função no PostgreSQL.

Exemplo de função:

```
create function pformat(band text, track text)
  returns text
as $$
  return '%s - %s' % (band, track)
$$ language plpythonu;
```

O código em Python foi marcado em verde.

Saída da função (através do psql):

```
music=> select pformat(track, band) from tracks;
        pformat
-------------------------
 Kashmir - Led Zeppelin
 Starless - King Crimson
 Yes - Siberian Khatru
 Genesis - Supper's Ready
(4 registros)
```

O ambiente de execução de Python no PostgreSQL provê o módulo plpy (importado automaticamente) que é uma abstração para o acesso aos recursos do SGBD.

Exemplo com plpy:

```
create function inibands()
```

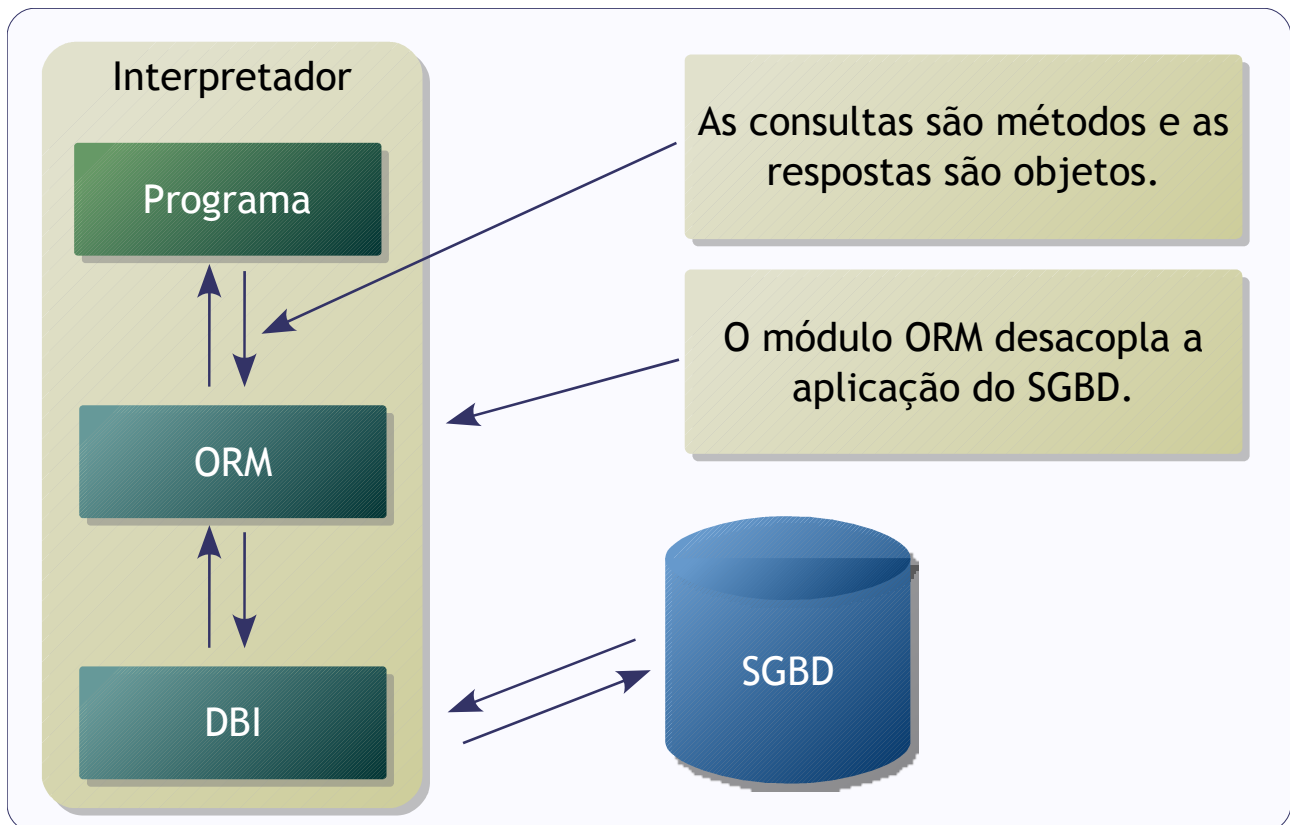
```
  returns setof text
as $$
  bands = plpy.execute('select distinct band from tracks order by 1')
  return [''.join(filter(lambda c: c == c.upper(), list(band['band']))) for band in bands]
$$ language plpythonu;
```

Saída da função (através do utilitário psql):

```
music=> select inibands();
 inibands
----------
 KC
 LZ
 Y
 G
(4 registros)
```

Funções Python pode ser utilizado tanto em *Stored Procedures* quanto *Triggers* no PostgreSQL.

Mapeamento objeto-relacional

Object-Relacional Mapper (ORM) é uma camada que se posiciona entre o código com a lógica da aplicação e o módulo DBI, com o objetivo de reduzir as dificuldades geradas pelas diferenças entre a representação de objetos (da linguagem) e a representação relacional (do banco de dados).

Com o uso de um ORM:

- A aplicação se torna independente do SGDB.
- O desenvolvedor não precisa usar SQL.
- A lógica para gerenciamento das conexões é realizada de forma transparente pelo ORM.

Exemplo de ORM (com SQLAlchemy[41]):

```
# -*- coding: latin1 -*-

# Testado com SQLAlchemy 0.44
from sqlalchemy import *
```

41 Documentação e fontes podem encontrados em: http://www.sqlalchemy.org/.

```
# URL => driver://username:password@host:port/database
# No SQLite:
#   sqlite:// (memória)
#   sqlite:///arquivo (arquivo em disco)
db = create_engine('sqlite:///progs.db')

# Torna acessível os metadados
metadata = MetaData(db)

# Ecoa o que SQLAlchemy está fazendo
metadata.bind.echo = True

# Tabela Prog
prog_table = Table('progs', metadata,
    Column('prog_id', Integer, primary_key=True),
    Column('name', String(80)))

# Cria a tabela
prog_table.create()

# Carrega a definição da tabela
prog_table = Table('progs', metadata, autoload=True)

# Insere dados
i = prog_table.insert()
i.execute({'name': 'Yes'}, {'name': 'Genesis'},
    {'name': 'Pink Floyd'}, {'name': 'King Crimson'})

# Seleciona
s = prog_table.select()
r = s.execute()

for row in r.fetchall():
    print row
```

Saída:

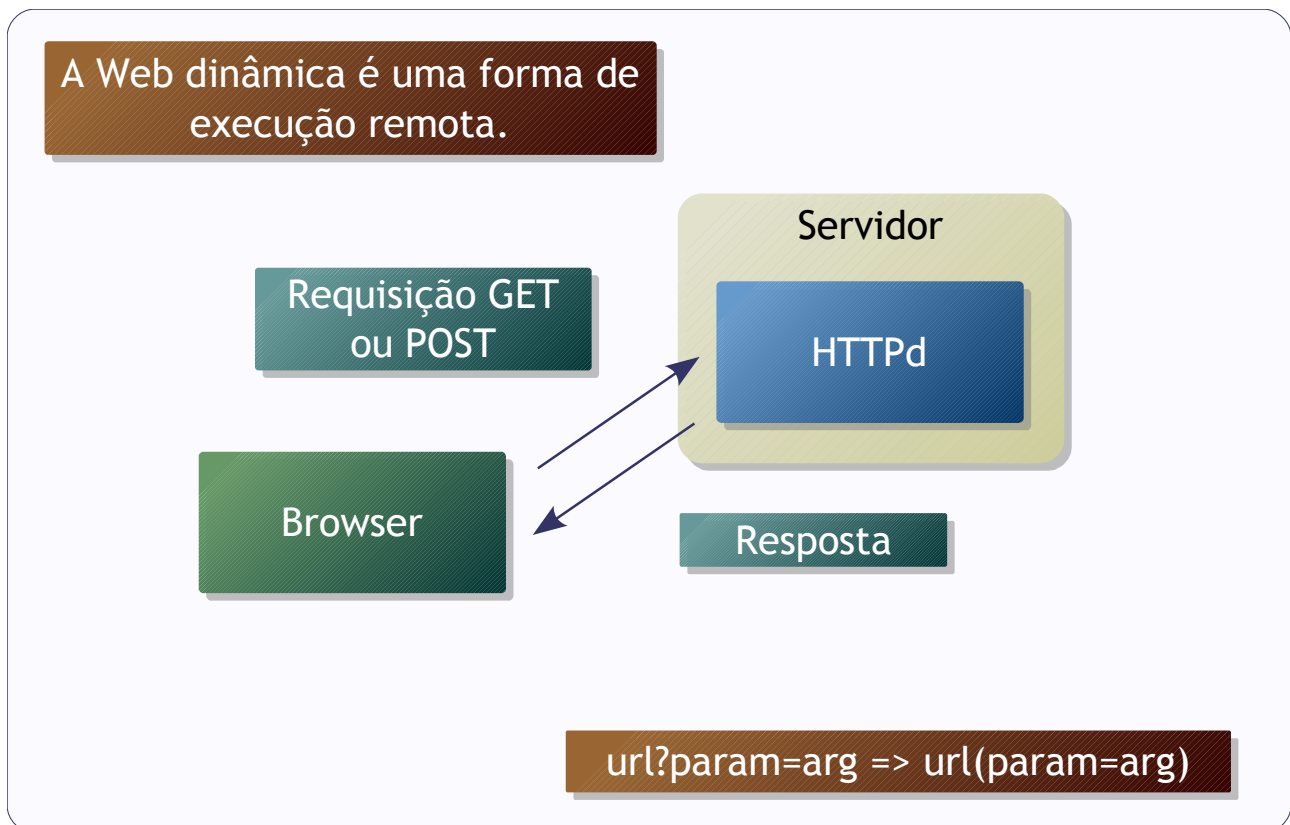
```
2008-05-04 10:50:35,068 INFO sqlalchemy.engine.base.Engine.0x..b0
CREATE TABLE progs (
        prog_id INTEGER NOT NULL,
        name VARCHAR(80),
        PRIMARY KEY (prog_id)
)


2008-05-04 10:50:35,069 INFO sqlalchemy.engine.base.Engine.0x..b0 {}
2008-05-04 10:50:38,252 INFO sqlalchemy.engine.base.Engine.0x..b0 COMMIT
2008-05-04 10:50:38,252 INFO sqlalchemy.engine.base.Engine.0x..b0 INSERT INTO progs
(name) VALUES (?)
```

```
2008-05-04 10:50:38,253 INFO sqlalchemy.engine.base.Engine.0x..b0 [['Yes'], ['Genesis'],
['Pink Floyd'], ['King Crimson']]
2008-05-04 10:50:38,267 INFO sqlalchemy.engine.base.Engine.0x..b0 COMMIT
2008-05-04 10:50:38,658 INFO sqlalchemy.engine.base.Engine.0x..b0 SELECT
progs.prog_id, progs.name
FROM progs
2008-05-04 10:50:38,660 INFO sqlalchemy.engine.base.Engine.0x..b0 []
(1, u'Yes')
(2, u'Genesis')
(3, u'Pink Floyd')
(4, u'King Crimson')
```

Além dos SQLAlchemy, também existem disponíveis para Python o SQLObject[42] e ORMs que integram *frameworks* maiores, como o Django.

42 Documentação e fontes disponíveis em: http://www.sqlobject.org/.

Web

Uma aplicação *Web* é uma aplicação cliente-servidor aonde o cliente é o *browser* (como o Mozilla Firefox) e o protocolo utilizado para a comunicação com o servidor é chamado *Hypertext Transfer Protocol* (HTTP), tecnologias que servem de base para a *World Wide Web* (WWW), as páginas de hipertexto que fazem parte da internet. Tais páginas seguem as convenções da linguagem *HyperText Markup Language*[43] (HTML).

As aplicações *Web* geram as páginas HTML dinamicamente, atendendo as requisições enviadas pelo *browser*. Aplicações *Web*, se construídas da forma adequada, podem ser acessadas em diversos ambientes além dos computadores pessoais, tais como PDAs e celulares.

Existem muitos *frameworks* para facilitar o desenvolvimento de aplicativos *Web* em Python, entre eles, o CherryPy e o CherryTemplate.

43 Especificações em: http://www.w3.org/MarkUp/.

CherryPy

CherryPy[44] é um *framework* para aplicações *Web* que publica objetos, convertendo URLs em chamadas para os métodos dos objetos publicados.

Exemplo com CherryPy:

```
import cherrypy


class Root(object):

    @cherrypy.expose
    def index(self):

        return 'Hello World!'

cherrypy.quickstart(Root())
```

O decorador *@expose* indica quais métodos são publicados via *Web*. O retorno do método é uma *string*, que é enviada para o *browser*.

O endereço padrão do servidor é "http://localhost:8080/".

CherryTemplate

CherryTemplate[45] é um módulo de processamento de modelos (*templates*) para Python. Era parte integrante do CherryPy, mas hoje é distribuído como um pacote separado.

Marcadores disponíveis no CherryTemplate:

- *py-eval*: avalia uma expressão em Python e insere o resultado (que deve ser uma *string*) no texto.

Exemplo:

```
Somatório de 1 a 10 é <py-eval="str(sum(range(1, 11)))">
```

- *py-exec*: executa uma linha de código Python.

Exemplo:

44 Documentação e fontes podem ser encontrados em: http://www.cherrypy.org/.
45 Documentação e fontes podem ser encontrados em: http://cherrytemplate.python-hosting.com/.

```
<py-exec="import platform">
O sistema é <py-eval="platform.platform()">
```

- *py-code*: executa um bloco de código Python.

Exemplo:

```
<py-code="
import platform
sistema = platform.platform()
">
<py-eval="sistema">
```

- *py-if* / *py-else*: funciona como o par *if* / *else* em Python.

Exemplo:

```
<py-if="1 > 10">
   Algo errado...
</py-if><py-else>
   Correto!
</py-else>
```

- *py-for*: funciona como o laço for em Python.

Exemplo:

```
<py-for="x in range(1, 11)">
   <py-eval="str(x)"> ** 2 = <py-eval="str(x ** 2)"><br>
</py-for>
```

- *py-include*: inclui um arquivo externo no *template*.

Exemplo:

```
<py-include="header.html">
Corpo da página...
<py-include="footer.html">
```

Além de usar uma *string* como *template*, é possível guardar o *template* em um arquivo:

```
renderTemplate(file='index.html')
```

Exemplo com *CherryTemplate*:

```
from cherrytemplate import renderTemplate

progs = ['Yes', 'Genesis', 'King Crimson']

template = '<html>\n<body>\n'\
'<py-for="prog in progs">'\
'    <py-eval="prog"><br>\n'\
'</py-for>'\
'</body>\n</html>\n'

print renderTemplate(template)
```

Saída HTML:

```
<html>
<body>
    Yes<br>
    Genesis<br>
    King Crimson<br>
</body>
</html>
```

As saídas geradas pelo CherryTemplate podem ser publicadas pelo CherryPy.

Cliente Web

O Python também pode funcionar do lado cliente, através do módulo *urllib*.

Exemplo:

```
# -*- coding: latin1 -*-

import urllib

# Abre a URL e fornece um objeto semelhante
# a um arquivo convencional
url = urllib.urlopen('http://ark4n.wordpress.com/')

# Lê a página
html = url.read()
```

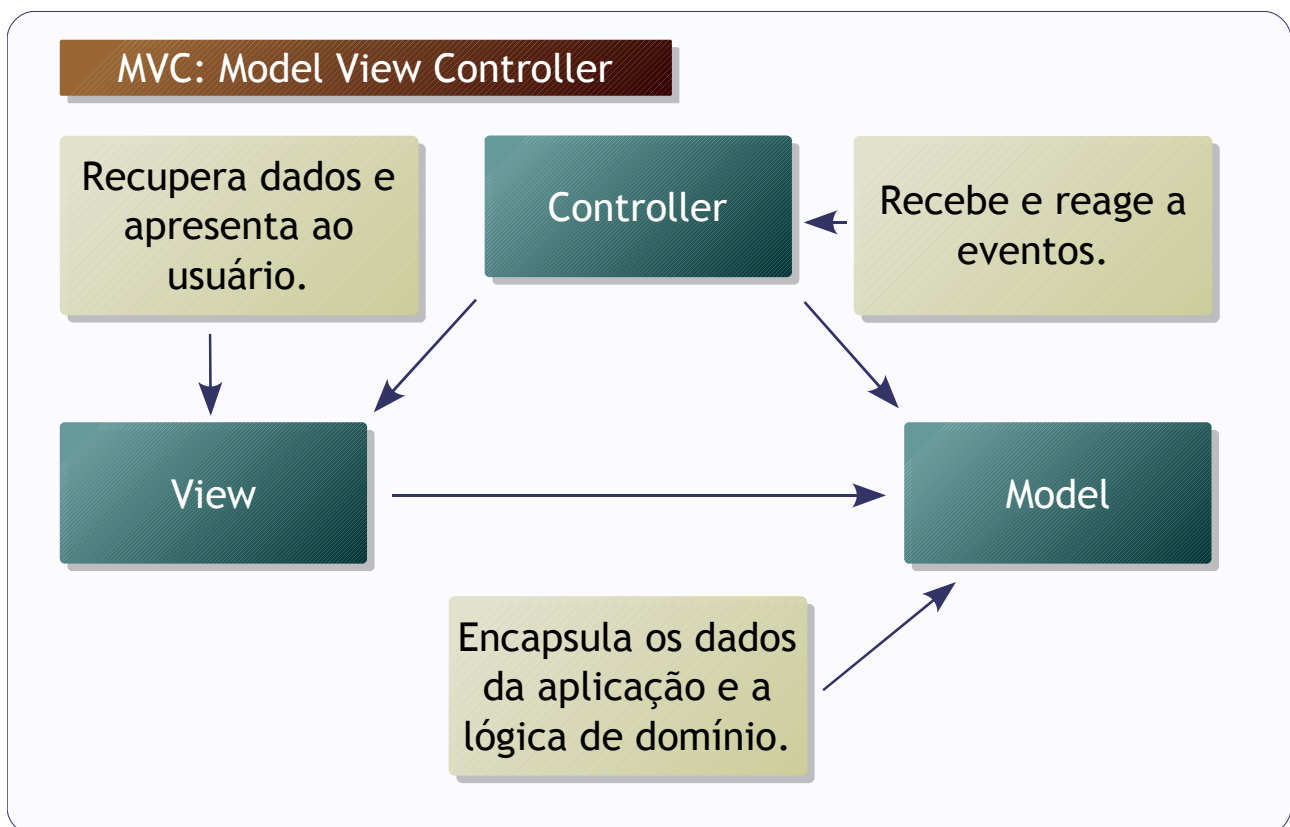
```
#html = '<a href="http://www.gnu.org/">'
found = html.find('href=', 0)

# find retorna -1 se não encontra
while found >= 0:

    # O fim do link (quando as aspas acabam)
    end = html.find(html[found + 5], found + 6) + 1

    # Mostra o link
    print html[found:end]

    # Passa para o próximo link
    found = html.find('href=', found + 1)
```

Outra solução cliente é o Twisted Web[46], que é parte do projeto Twisted[47], um *framework* orientado a eventos voltado para protocolos de rede, incluindo HTTP, SSH, IRC, IMAP e outros.

46 Endereço: http://twistedmatrix.com/trac/wiki/TwistedWeb.
47 Endereço: http://twistedmatrix.com/trac/.

MVC

Model-view-controller (MVC) é uma arquitetura de software que divide a aplicação em três partes distintas: o modelo de dados da aplicação, a interface com o usuário e a lógica de controle.

O objetivo é obter um baixo acoplamento entre as três partes de forma que uma alteração em uma parte tenha pouco (ou nenhum) impacto nas outras partes.

A criação da aplicação dependente da definição de três componentes:

- Modelo (*model*): encapsula os dados da aplicação e a lógica de domínio.
- Visão (*view*): recupera dados do modelo e apresenta ao usuário.
- Controlador (*controller*): recebe e reage a eventos, como interações com o usuário e requisita alterações no modelo e na visão.

Embora a arquitetura não determine formalmente a presença de um componente de persistência, fica implícito que este faz parte do componente modelo.

O uso mais comum para o modelo MVC é em aplicações *Web* baseadas em bancos de dados, que implementam as operações básicas chamadas CRUD (*Create, Read, Update and*

Delete).

Existem vários *frameworks* para aumentar a produtividade na criação de aplicativos seguindo o MVC, com recursos como:

- *Scripts* que automatizam as tarefas mais comuns de desenvolvimento.
- Geração automática de código.
- Uso de ORM.
- Uso de CSS[48] (*Cascade Style Sheets*).
- Uso de AJAX (*Asynchronous Javascript And XML*).
- Modelos de aplicações.
- Uso de introspecção para obter informações sobre as estruturas de dados e gerar formulários com campos com as características correspondentes.
- Diversas opções pré-configuradas com *defaults* adequados para a maioria das aplicações.

O *framework* MVC mais conhecido é o Ruby On Rails[49], que ajudou a popularizar o MVC entre os desenvolvedores.

Especificamente desenvolvidos em Python, existem os *frameworks* Django[50] e TurboGears[51], entre outros.

Exemplo:

```
# -*- coding: utf-8 -*-
"""
Web com operações CRUD
"""

# CherryPy
import cherrypy

# CherryTemplate
import cherrytemplate

# SQLAlchemy
import sqlalchemy as sql

# Conecta ao bando
db = sql.create_engine('sqlite:///zoo.db')
```

48 Página oficial em: http://www.w3.org/Style/CSS/.
49 Página oficial em: http://www.rubyonrails.org/.
50 Página oficial em: http://www.djangoproject.com/.
51 Página oficial em: http://turbogears.org/.

```
# Acesso aos metadados
metadata = sql.MetaData(db)

try:
    # Carrega metadados da tabela
    zoo = sql.Table('zoo', metadata, autoload=True)

except:
    # Define a estrutura da tabela zoo
    zoo = sql.Table('zoo', metadata,
        sql.Column('id', sql.Integer, primary_key=True),
        sql.Column('nome', sql.String(100),
            unique=True, nullable=False),
        sql.Column('quantidade', sql.Integer, default=1),
        sql.Column('obs', sql.String(200), default='')
    )

    # Cria a tabela
    zoo.create()

# Os nomes das colunas
colunas = [col for col in zoo.columns.keys()]
colunas.remove('id')


class Root(object):
    """Raiz do site"""

    @cherrypy.expose
    def index(self, **args):
        """
        Lista os registros
        """

        msg = ''
        op = args.get('op')
        ident = int(args.get('ident', 0))
        novo = {}

        for coluna in colunas:
            novo[coluna] = args.get(coluna)

        if op == 'rem':

            # Remove dados
            rem = zoo.delete(zoo.c.id==ident)
            rem.execute()
            msg = 'registro removido.'

        elif op == 'add':

            novo = {}
```

```
            for coluna in colunas:
                novo[coluna] = args[coluna]

            try:
                # Insere dados
                ins = zoo.insert()
                ins.execute(novo)
                msg = 'registro adicionado.'

            except sql.exceptions.IntegrityError:
                msg = 'registro existe.'

        elif op == 'mod':

            novo = {}

            for coluna in colunas:
                novo[coluna] = args[coluna]

            try:
                # Modifica dados
                mod = zoo.update(zoo.c.id==ident)
                mod.execute(novo)
                msg = 'registro modificado.'

            except sql.exceptions.IntegrityError:
                msg = 'registro existe.'

        # Seleciona dados
        sel = zoo.select(order_by=zoo.c.nome)
        rec = sel.execute()

        # Gera a página principal a partir do modelo "index.html"
        return cherrytemplate.renderTemplate(file='index.html',
            outputEncoding='utf-8')

    @cherrypy.expose
    def add(self):
        """
        Cadastra novos registros
        """

        # Gera a página de registro novo a partir do modelo "add.html"
        return cherrytemplate.renderTemplate(file='add.html',
            outputEncoding='utf-8')

    @cherrypy.expose
    def rem(self, ident):
        """
        Confirma a remoção de registros
        """
```

```
        # Seleciona o registro
        sel = zoo.select(zoo.c.id==ident)
        rec = sel.execute()
        res = rec.fetchone()

        # Gera a página de confirmar exclusão a partir do modelo "rem.html"
        return cherrytemplate.renderTemplate(file='rem.html',
            outputEncoding='utf-8')

    @cherrypy.expose
    def mod(self, ident):
        """
        Modifica registros
        """

        # Seleciona o registro
        sel = zoo.select(zoo.c.id==ident)
        rec = sel.execute()
        res = rec.fetchone()

        # Gera a página de alteração de registro a partir do modelo "mod.html"
        return cherrytemplate.renderTemplate(file='mod.html',
            outputEncoding='utf-8')


# Inicia o servidor na porta 8080
cherrypy.quickstart(Root())
```

Modelo “index.html” (página principal):

```
<py-include="header.html">
<table>
<tr>
<th></th>
<py-for="coluna in colunas">
    <th><py-eval="coluna"></th>
</py-for>
<th></th>
<th></th>
</tr>
<py-for="i, campos in enumerate(rec.fetchall())">
    <tr>
    <th><py-eval="unicode(i + 1)"></th>
    <py-for="coluna in colunas">
        <td><py-eval="unicode(campos[coluna])"></td>
    </py-for>
    <td>
    <a href="/mod?ident=<py-eval="unicode(campos['id'])">">modificar</a>
    </td><td>
    <a href="/rem?ident=<py-eval="unicode(campos['id'])">">remover</a>
    </td>
    </tr>
</py-for>
```

```
</table>
<br />
<form action="/add" method="post">
    <input type="submit" value=" adicionar " />
</form>
<p>
<py-eval="msg">
</p>
<py-include="footer.html">
```

Modelo “add.html” (página de formulário para novos registros):

```
<py-include="header.html">
<form action="/?op=add" method="post">
   <table>
   <py-for="coluna in colunas">
      <tr><td>
      <py-eval="coluna">
      </td><td>
      <input type="text" size="30" name="<py-eval="coluna">" />
      </td></tr>
   </py-for>
   </table>
<br />
<input type="submit" value=" salvar " />
</form>
<br />
[ <a href="/">voltar</a> ]
<py-include="footer.html">
```

Modelo “mod.html” (página de formulário para alteração de registros):

```
<py-include="header.html">
<form action="/?op=mod&ident=<py-eval="unicode(res['id'])">" method="post">
   <table border="0">
   <py-for="coluna in colunas">
      <tr><th>
      <py-eval="coluna">
      </th><td>
      <input type="text" size="30" name="<py-eval="coluna">"
      value="<py-eval="unicode(res[coluna])">" />
      </td></tr>
   </py-for>
   </table>
<br />
<input type="submit" value=" salvar " />
</form>
<br />
[ <a href="/">voltar</a> ]
```

```
<py-include="footer.html">
```

Modelo “rem.html” (página que pede confirmação para remoção de registros):

```
<py-include="header.html">
<table border="1">
<tr>
<py-for="coluna in colunas">
   <th><py-eval="coluna"></th>
</py-for>
</tr>
<tr>
<py-for="coluna in colunas">
   <td><py-eval="unicode(res[coluna])"></td>
</py-for>
</tr>
</table>
<br />
<form action="/?op=rem&ident=<py-eval="unicode(res['id'])">" method="post">
   <input type="submit" value=" remover " />
</form>
<br />
[ <a href="/">voltar</a> ]
<py-include="footer.html">
```

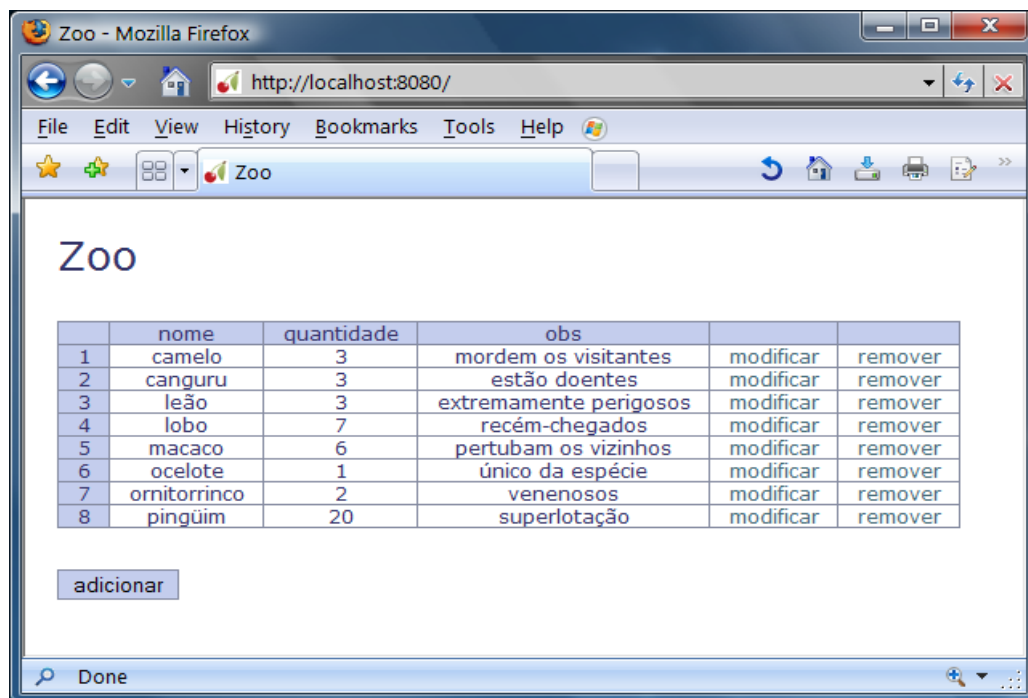
Modelo “header.html” (cabeçalho comum a todos os modelos):

```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
<title>Zoo</title>
<style type="text/css">
<!--
body {
   margin: 10;
   padding: 10;
   font: 80% Verdana, Lucida, sans-serif;
   color: #333366;
}
h1 {
   margin: 0;
   padding: 0;
   font: 200% Lucida, Verdana, sans-serif;
}
a {
   color: #436976;
   text-decoration: none;
}
```

```
a:hover {
   background: #c4cded;
   text-decoration: underline;
}
table {
   margin: 1em 0em 1em 0em;
   border-collapse: collapse;
   border-left: 1px solid #858ba1;
   border-bottom: 1px solid #858ba1;
   font: 90% Verdana, Lucida, sans-serif;
}
table th {
   padding: 0em 1em 0em 1em;
   border-top: 1px solid #858ba1;
   border-bottom: 1px solid #858ba1;
   border-right: 1px solid #858ba1;
   background: #c4cded;
   font-weight: normal;
}
table td {
   padding: 0em 1em 0em 1em;
   border-top: 1px solid #858ba1;
   border-right: 1px solid #858ba1;
   text-align: center;
}
form {
   margin: 0;
   border: none;
}
input {
   border: 1px solid #858ba1;
   background-color: #c4cded;
   vertical-align: middle;
}
-->
</style>
</head>
<body>
<h1>Zoo</h1>
<br />
```

Modelo “footer.html” (rodapé comum a todos os modelos):

```
</body>
</html>
```

Página principal:

Zoo - Mozilla Firefox
http://localhost:8080/
File Edit View History Bookmarks Tools Help
Zoo
Zoo
nome quantidade obs
1 camelo 3 mordem os visitantes modificar remover
2 canguru 3 estão doentes modificar remover
3 leão 3 extremamente perigosos modificar remover
4 lobo 7 recém-chegados modificar remover
5 macaco 6 pertubam os vizinhos modificar remover
6 ocelote 1 único da espécie modificar remover
7 ornitorrinco 2 venenosos modificar remover
8 pingüim 20 superlotação modificar remover
adicionar
Done

Exercícios V

1. Implementar uma classe *Animal* com os atributos: nome, espécie, gênero, peso, altura e idade. O objeto derivado desta classe deverá salvar seu estado em arquivo com um método chamado "salvar" e recarregar o estado em um método chamado "desfazer".

2. Implementar uma função que formate uma lista de tuplas como tabela HTML.

3. Implementar uma aplicação *Web* com uma saudação dependente do horário (exemplos: "Bom dia, são 09:00.", "Boa tarde, são 13:00." e "Boa noite, são 23:00.").

4. Implementar uma aplicação *Web* com um formulário que receba expressões Python e retorne a expressão com seu resultado.

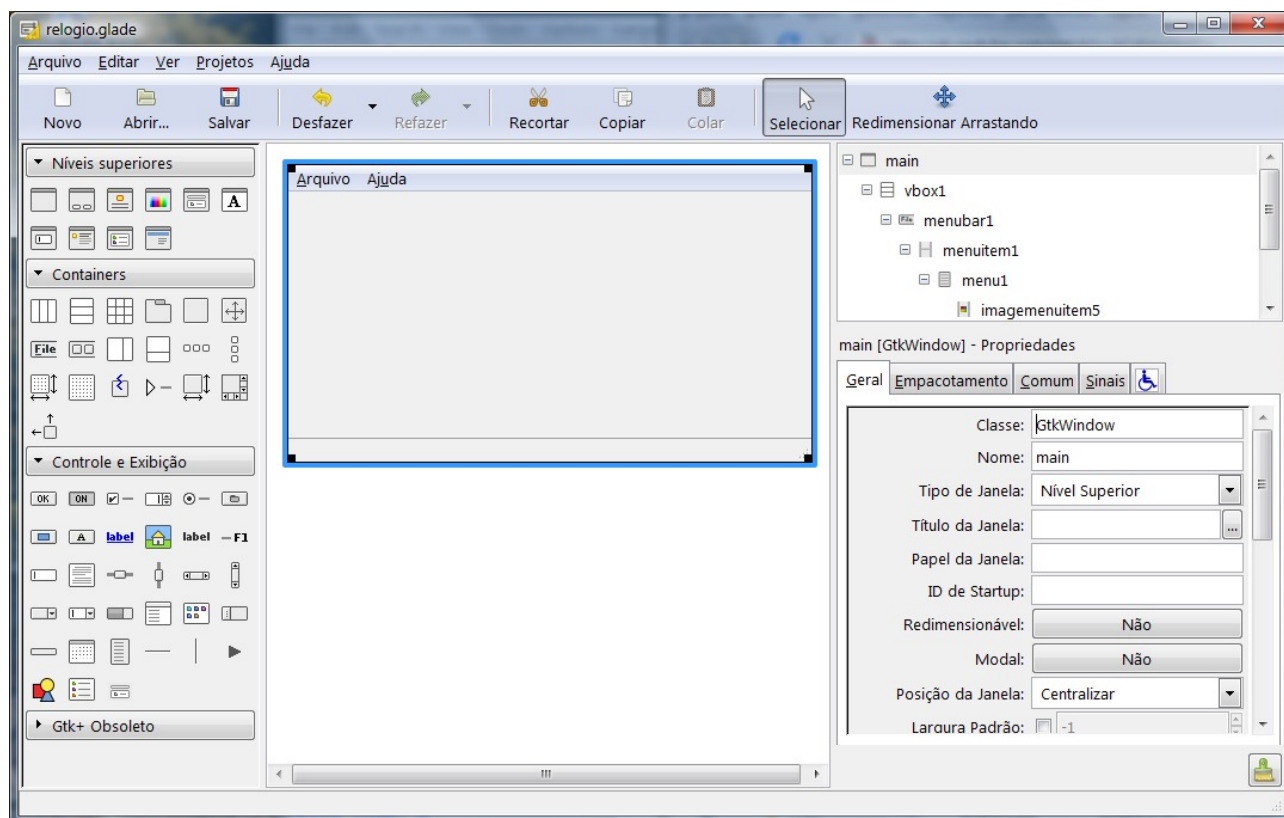
Parte VI

- Interface gráfica.
- Threads.
- Processamento distribuído.
- Performance.
- Exercícios VI.

Interface Gráfica

As Interfaces Gráficas com Usuário (GUI, *Graphic User Interface*) se popularizaram no ambiente *desktop*, devido à facilidade de uso e a produtividade. Existem hoje muitas bibliotecas disponíveis para a construção de aplicações GUI, tais como: GTK+, QT, TK e wxWidgets.

O GTK+[52] (GIMP Toolkit) é uma biblioteca *Open Source* escrita em linguagem C. Originalmente concebida para ser usada pelo GIMP[53], é compatível com as plataformas mais utilizadas atualmente e rica em recursos, entre eles, um construtor de interfaces chamado Glade.

Interface do Glade:

O GTK+ é usado pelo GNOME[54] (ambiente *desktop Open Source*) e por diversas aplicativos, como os portes do Mozilla Firefox e do BrOffice.org para sistemas UNIX. O GTK+ pode ser

52 A página internet do projeto reside em: http://www.gtk.org/. e os binários para Windows estão disponíveis em: http://gladewin32.sourceforge.net/. A versão para desenvolvedores instala o Glade.
53 Endereço oficial do projeto: http://www.gimp.org/.
54 Documentação e fontes em: http://www.gnome.org/.

usado no Python através do pacote PyGTK[55]. Os portes das bibliotecas para Windows podem ser encontrados em:

- PyGTK: http://ftp.gnome.org/pub/gnome/binaries/win32/pygtk/
- PyGObject: http://ftp.gnome.org/pub/gnome/binaries/win32/pygobject/
- PyCairo: http://ftp.gnome.org/pub/gnome/binaries/win32/pycairo/

Arquitetura

Interfaces gráficas geralmente utilizam a metáfora do *desktop*, um espaço em duas dimensões, é que ocupado por janelas retangulares, que representam aplicativos, propriedades ou documentos.

As janelas podem conter diversos tipos de controles (objetos utilizados para interagir com o usuário ou para apresentar informações) e *containers* (objetos que servem de repositório para coleções de outros objetos).

A interface gráfica deve ficar esperando por eventos e responder de acordo. Os eventos podem ser resultado da interação do usuário, como cliques e arrastar de mouse ou digitação ou de eventos de sistema, como o relógio da máquina. A reação a um evento qualquer é definida através de funções *callback* (funções que são passadas como argumento para outras rotinas).

Controles mais usados:

- Rótulo (*label*): retângulo que exibe texto.
- Caixa de texto (*text box*): área de edição de texto.
- Botão (*button*): área que pode ser ativada interativamente.
- Botão de rádio (*radio button*): tipo especial de botão, com o qual são formados grupos aonde apenas um pode ser apertado de cada vez.
- Botão de verificação (*check button*): botão que pode ser marcado e desmarcado.
- Barras de rolagem (*scroll bars*): controles deslizantes horizontais ou verticais, usados para intervalos de valores numéricos.
- Botão giratório (*spin button*): caixa de texto com dois botões com setas ao lado que incrementam e decrementam o número na caixa.
- Barra de status (*status bar*): barra para exibição de mensagens, geralmente na parte inferior da janela.
- Imagem (*image*): área para exibição de imagens.

Controles podem ter aceleradores (teclas de atalho) associados a eles.

55 A página na internet do PyGTK é http://www.pygtk.org/.

Containers mais usados:

- Barra de menu (*menu bar*): sistema de menus, geralmente na parte superior da janela.
- Fixo (*fixed*): os objetos ficam fixados nas mesmas posições.
- Tabela (*table*): coleção de compartimentos para fixar os objetos, distribuídos em linhas e colunas.
- Caixa horizontal (*horizontal box*): semelhante à tabela, porém apenas com uma linha.
- Caixa vertical (*vertical box*): semelhante à tabela, porém apenas com uma coluna.
- Caderno (*notebook*): várias áreas que podem ser visualizadas uma cada vez quando selecionadas através de abas, geralmente na parte superior.
- Barra de ferramentas (*tool bar*): área com botões para acesso rápido aos principais recursos do aplicativo.

Para lidar com eventos de tempo, as interfaces gráficas implementam um recurso chamado temporizador (*timer*) que evoca a função *callback* depois de um certo tempo programado.

Construindo interfaces

Embora seja possível criar interfaces inteiramente usando código, é mais produtivo construir a interface em um software apropriado. O Glade gera arquivos XML com extensão ".glade", que podem ser lidos por programas que usam GTK+, automatizando o processo de criar interfaces.

Roteiro básico para construir uma interface:

No Glade:

- Crie uma janela usando algum dos modelos disponíveis em "Níveis Superiores".
- Crie *containers* para armazenar os controles.
- Crie os controles.
- Crie os manipuladores para os sinais necessários.
- Salve o arquivo com a extensão ".glade".

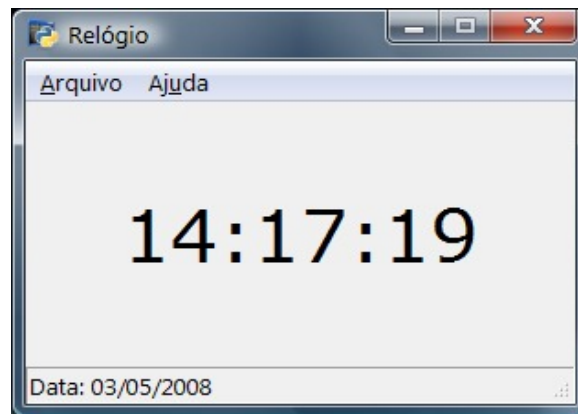
No Python:

- Importe os pacotes necessários.
- Use o GTK para interpretar o arquivo XML do Glade.
- Crie rotinas para serem usadas como funções *callback*.
- Associe as rotinas com os manipuladores correspondes que foram criados no Glade, através do método *signal_autoconnect()*.
- Ative o laço para processar eventos com *gtk.main()*.

Exemplo (relógio):

No Glade:

- Clique em "janela" em "Níveis Superiores".
- Nas propriedades da janela:
  - Mude "Nome" para "main" em "Geral".
  - Mude "Redimensionável" para "Sim".
  - Mude "Posição da janela" para "Centralizar".
  - Mude "Visível" para "Sim" em "Comum".
  - Mude o manipulador para "on_main_destroy" do sinal "destroy" de "GtkObject" em "Sinais".
- Clique em "Caixa vertical" em "Containers", depois clique dentro da janela e escolha o número de itens igual a 3.
- Clique em "Barra de menu" em "Containers", depois clique dentro do espaço vazio superior e delete os itens "Editar" e "Ver".
- Clique em "Barra de status" em "Controle e Exibição" e depois clique dentro do espaço vazio inferior.
- Mude o nome da barra de status para "sts_data" em "Geral".
- Clique em "Rótulo" em "Controle e Exibição" e depois clique dentro do espaço vazio central.
- Nas propriedades do rótulo, mude "Nome" para "lbl_hora" e "Rótulo" para vazio em "Geral", "Solicitação de largura" para "300" e "Solicitação de altura" para "150" em "Comum".
- No "Inspetor" (lista em forma de árvore com todos itens), delete:
  - "imagemenuitem1".
  - "imagemenuitem2".
  - "imagemenuitem3".
  - "imagemenuitem4".
  - "separatormenuitem1".
- No "Inspetor":
  - localize "imagemenuitem5" e mude o manipulador em "Sinais" do sinal "activate" para "on_imagemenuitem5_activate" de "GtkMenuItem".
  - localize "imagemenuitem10" e mude o manipulador em "Sinais" do sinal "activate" para "on_imagemenuitem10_activate" de "GtkMenuItem".
- Salve o arquivo como "relogio.glade".

Janela principal do relógio:

Código em Python:

```
# -*- coding: latin1 -*-
"""
Um relógio com GTK.
"""

import datetime

# GTK e outros módulos associados
import gtk
import gtk.glade
import gobject
import pango


class Relogio(object):
    """
    Implementa a janela principal do programa.
    """

    def __init__(self):
        """
        Inicializa a classe.
        """

        # Carrega a interface
        self.tree = gtk.glade.XML('relogio.glade', 'main')

        # Liga os eventos
        callbacks = {
            'on_main_destroy': self.on_main_destroy,
            'on_imagemenuitem5_activate': self.on_main_destroy,
            'on_imagemenuitem10_activate': self.on_imagemenuitem10_activate
            }

        self.tree.signal_autoconnect(callbacks)
```

```
        # Coloca um título na janela
        self.tree.get_widget('main').set_title('Relógio')

        # O rótulo que reberá a hora
        self.hora = self.tree.get_widget('lbl_hora')

        # A barra de status que reberá a data
        self.data = self.tree.get_widget('sts_data')
        print dir(self.data)

        # Muda a fonte do rótulo
        self.hora.modify_font(pango.FontDescription('verdana 28'))

        # Um temporizador para manter a hora atualizada
        self.timer = gobject.timeout_add(1000, self.on_timer)

    def on_imagemenuitem10_activate(self, widget):
        """
        Cria a janela de "Sobre".
        """

        # Caixa de dialogo
        dialog = gtk.MessageDialog(parent=self.tree.get_widget('main'),
            flags=gtk.DIALOG_MODAL | gtk.DIALOG_DESTROY_WITH_PARENT,
            type=gtk.MESSAGE_OTHER, buttons=gtk.BUTTONS_OK,
            message_format='Primeiro exemplo usando GTK.')

        dialog.set_title('Sobre')
        dialog.set_position(gtk.WIN_POS_CENTER_ALWAYS)

        # Exibe a caixa
        dialog.run()
        dialog.destroy()
        return

    def on_timer(self):
        """
        Rotina para o temporizador.
        """

        # Pega a hora do sistema
        hora = datetime.datetime.now().time().isoformat().split('.')[0]

        # Muda o texto do rótulo
        self.hora.set_text(hora)

        # Pega a data do sistema em formato ISO
        data = datetime.datetime.now().date().isoformat()
        data = 'Data: ' + '/'.join(data.split('-')[::-1])

        # Coloca a data na barra de status
        self.data.push(0, data)
```

```
        # Verdadeiro faz com que o temporizador rode de novo
        return True

    def on_main_destroy(self, widget):
        """
        Termina o programa.
        """

        raise SystemExit


if __name__ == "__main__":

    # Inicia a GUI
    relogio = Relogio()
    gtk.main()
```

Arquivo “relogio.glade”:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.3 on Sat May 03 14:06:18 2008 -->
<glade-interface>
  <widget class="GtkWindow" id="main">
    <property name="visible">True</property>
    <property name="resizable">False</property>
    <property name="window_position">GTK_WIN_POS_CENTER</property>
    <signal name="destroy" handler="on_main_destroy"/>
    <child>
      <widget class="GtkVBox" id="vbox1">
        <property name="visible">True</property>
        <child>
          <widget class="GtkMenuBar" id="menubar1">
            <property name="visible">True</property>
            <child>
              <widget class="GtkMenuItem" id="menuitem1">
                <property name="visible">True</property>
                <property name="label" translatable="yes">_Arquivo</property>
                <property name="use_underline">True</property>
                <child>
                  <widget class="GtkMenu" id="menu1">
                    <property name="visible">True</property>
                    <child>
                      <widget class="GtkImageMenuItem" id="imagemenuitem5">
                        <property name="visible">True</property>
                        <property name="label" translatable="yes">gtk-quit</property>
                        <property name="use_underline">True</property>
                        <property name="use_stock">True</property>
                        <signal name="activate" handler="on_imagemenuitem5_activate"/>
```

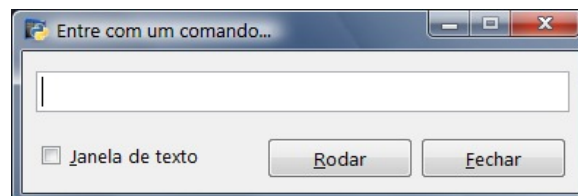
```
            </widget>
          </child>
        </widget>
      </child>
    </widget>
  </child>
  <child>
    <widget class="GtkMenuItem" id="menuitem4">
      <property name="visible">True</property>
      <property name="label" translatable="yes">Aj_uda</property>
      <property name="use_underline">True</property>
      <child>
        <widget class="GtkMenu" id="menu3">
          <property name="visible">True</property>
          <child>
            <widget class="GtkImageMenuItem" id="imagemenuitem10">
              <property name="visible">True</property>
              <property name="label" translatable="yes">gtk-about</property>
              <property name="use_underline">True</property>
              <property name="use_stock">True</property>
              <signal name="activate" handler="on_imagemenuitem10_activate"/>
            </widget>
          </child>
        </widget>
      </child>
    </widget>
  </child>
</widget>
<packing>
  <property name="expand">False</property>
</packing>
</child>
<child>
  <widget class="GtkLabel" id="lbl_hora">
    <property name="width_request">300</property>
    <property name="height_request">150</property>
    <property name="visible">True</property>
    <property name="xpad">5</property>
    <property name="ypad">5</property>
  </widget>
  <packing>
    <property name="position">1</property>
  </packing>
</child>
<child>
  <widget class="GtkStatusbar" id="sts_data">
    <property name="visible">True</property>
    <property name="spacing">2</property>
  </widget>
  <packing>
    <property name="expand">False</property>
    <property name="position">2</property>
```

```
      </packing>
     </child>
    </widget>
   </child>
  </widget>
</glade-interface>
```

Exemplo (rodando programas):

No Glade:

- Crie uma janela com o nome “main” com o manipulador “on_main_destroy” para o sinal “destroy”.
- Crie um *container* fixo para receber os controles.
- Crie uma caixa de texto chamada “ntr_cmd”. Esta caixa receberá comandos para serem executados.
- Crie um botão de verificação chamado “chk_shell”, com o texto “Janela de texto”. Se o botão estiver marcado, o comando será executado em uma janela de texto.
- Crie um botão chamado “btn_rodar” com o manipulador “on_btn_fechar_clicked” para o sinal “clicked”. Quando clicado, o comando da caixa de texto é executado.
- Crie um botão chamado “btn_fechar” com o manipulador “on_btn_fechar_clicked” para o sinal “clicked”. Quando clicado, o programa termina.

Janela principal:

Código em Python:

```
# -*- coding: utf-8 -*-
"""
Rodando programas com GTK.
"""

import subprocess

import gtk
import gtk.glade
import gobject
import pango
```

```
class Exec(object):
    """
    Janela principal.
    """

    def __init__(self):
        """
        Inicializa a classe.
        """

        # Carrega a interface
        self.tree = gtk.glade.XML('cmd.glade', 'main')

        # Liga os eventos
        callbacks = {
            'on_main_destroy': self.on_main_destroy,
            'on_btn_fechar_clicked': self.on_main_destroy,
            'on_btn_rodar_clicked': self.on_btn_rodar_clicked
            }

        self.tree.signal_autoconnect(callbacks)

    def on_btn_rodar_clicked(self, widget):
        """
        Roda o comando.
        """

        ntr_cmd = self.tree.get_widget('ntr_cmd')
        chk_shell = self.tree.get_widget('chk_shell')

        cmd = ntr_cmd.get_text()
        if cmd:
            # chk_shell.state Ã© 1 se chk_shell estiver marcado
            if chk_shell.state:
                cmd = 'cmd start cmd /c ' + cmd
            subprocess.Popen(args=cmd)

        else:
            # Caixa de dialogo
            dialog = gtk.MessageDialog(parent=self.tree.get_widget('main'),
                flags=gtk.DIALOG_MODAL | gtk.DIALOG_DESTROY_WITH_PARENT,
                type=gtk.MESSAGE_OTHER, buttons=gtk.BUTTONS_OK,
                message_format='Digite um comando.')

            dialog.set_title('Mensagem')
            dialog.set_position(gtk.WIN_POS_CENTER_ALWAYS)

            # Exibe a caixa
            dialog.run()
            dialog.destroy()
```

```
        return True

    def on_main_destroy(self, widget):
        """
        Termina o programa.
        """

        raise SystemExit


if __name__ == "__main__":

    # Inicia a GUI
    exe = Exec()
    gtk.main()
```

O arquivo “cmd.glade”:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.3 on Tue May 27 23:44:03 2008 -->
<glade-interface>
  <widget class="GtkWindow" id="main">
    <property name="width_request">380</property>
    <property name="height_request">100</property>
    <property name="visible">True</property>
    <property name="title" translatable="yes">Entre com um comando...</property>
    <property name="resizable">False</property>
    <property name="modal">True</property>
    <property name="window_position">GTK_WIN_POS_CENTER</property>
    <signal name="destroy" handler="on_main_destroy"/>
    <child>
      <widget class="GtkFixed" id="fixed1">
        <property name="width_request">380</property>
        <property name="height_request">100</property>
        <property name="visible">True</property>
        <child>
          <widget class="GtkButton" id="btn_rodar">
            <property name="width_request">100</property>
            <property name="height_request">29</property>
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <property name="label" translatable="yes">_Rodar</property>
            <property name="use_underline">True</property>
            <property name="response_id">0</property>
            <signal name="clicked" handler="on_btn_rodar_clicked"/>
          </widget>
          <packing>
```

```
        <property name="x">167</property>
        <property name="y">61</property>
      </packing>
    </child>
    <child>
      <widget class="GtkButton" id="btn_fechar">
        <property name="width_request">100</property>
        <property name="height_request">29</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="receives_default">True</property>
        <property name="label" translatable="yes">_Fechar</property>
        <property name="use_underline">True</property>
        <property name="response_id">0</property>
        <signal name="clicked" handler="on_btn_fechar_clicked"/>
      </widget>
      <packing>
        <property name="x">272</property>
        <property name="y">61</property>
      </packing>
    </child>
    <child>
      <widget class="GtkEntry" id="ntr_cmd">
        <property name="width_request">365</property>
        <property name="height_request">29</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
      </widget>
      <packing>
        <property name="x">9</property>
        <property name="y">14</property>
      </packing>
    </child>
    <child>
      <widget class="GtkCheckButton" id="chk_shell">
        <property name="width_request">136</property>
        <property name="height_request">29</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="label" translatable="yes">_Janela de texto</property>
        <property name="use_underline">True</property>
        <property name="response_id">0</property>
        <property name="draw_indicator">True</property>
      </widget>
      <packing>
        <property name="x">11</property>
        <property name="y">59</property>
      </packing>
    </child>
  </widget>
 </child>
</widget>
```

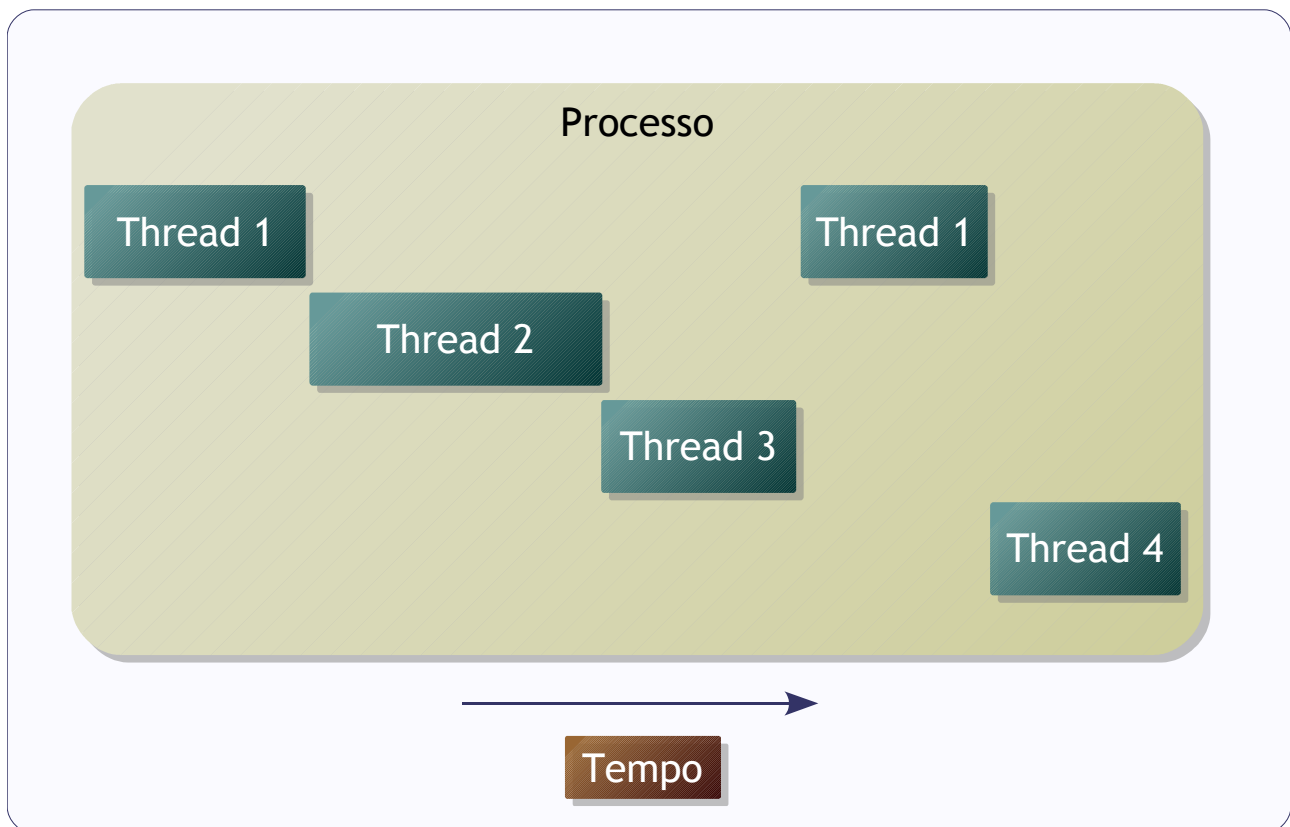
```
</glade-interface>
```

Além do Glade, também existe o Gaspacho[56], outro construtor de interfaces que também gera arquivos XML no padrão do Glade.

Funcionalidades associadas a interface gráfica podem ser obtidas usando outros módulos, como o pySystray[57], que implementa a funcionalidade que permite que o aplicativo use a bandeja de sistema no Windows.

56 Disponível em: http://gazpacho.sicem.biz/.
57 Endereço na internet: http://datavibe.net/~essiene/pysystray/.

Threads

Uma *thread* é uma linha de execução que compartilha sua área de memória com outras linhas, ao contrário do processo tradicional, que possui apenas uma linha com área de memória própria.

O uso de *threads* oferece algumas vantagens em relação aos processos convencionais:

- Consomem menos recursos de máquina.
- Podem ser criadas e destruídas mais rapidamente.
- Podem ser chaveadas mais rapidamente.
- Podem se comunicar com outras *threads* de forma mais fácil.

É comum utilizar *threads* para:

- Processamento paralelo, em casos como atender várias conexões em processos servidores.
- Executar operações de I/O assíncronas, por exemplo: enquanto o usuário continua interagindo com a interface enquanto a aplicação envia um documento para a impressora.
- Operações de I/O em paralelo.

Em Python, o módulo da biblioteca padrão *threading* provê classes de alto nível de abstração e usa o módulo *thread*, que implementa as rotinas de baixo nível e que geralmente não é usado diretamente.

Exemplo com o módulo *threading*:

```
# -*- coding: latin1 -*-
"""
Exemplo de uso de threads
"""

import os
import time
import threading


class Monitor(threading.Thread):
    """
    Classe de monitoramento usando threads
    """
    def __init__(self, ip):
        """
        Construtor da thread
        """
        # Atributos para a thread
        self.ip = ip
        self.status = None

        # Inicializador da classe Thread
        threading.Thread.__init__(self)

    def run(self):
        """
        Código que será executado pela thread
        """
        # Execute o ping
        ping = os.popen('ping -n 1 %s' % self.ip).read()

        if 'Esgotado' in ping:
            self.status = False
        else:
            self.status = True


if __name__ == '__main__':

    # Crie uma lista com um objeto de thread para cada IP
    monitores = []
    for i in range(1, 11):

        ip = '10.10.10.%d' % i
```

```
        monitores.append(Monitor(ip))

    # Execute as Threads
    for monitor in monitores:
        monitor.start()

    # A thread principal continua enquanto
    # as outras threads executam o ping
    # para os endereços da lista

    # Verifique a cada segundo
    # se as threads acabaram
    ping = True

    while ping:
        ping = False

        for monitor in monitores:
            if monitor.status == None:
                ping = True
                break

        time.sleep(1)

    # Imprima os resultados no final
    for monitor in monitores:

        if monitor.status:
            print '%s no ar' % monitor.ip
        else:
            print '%s fora do ar' % monitor.ip
```

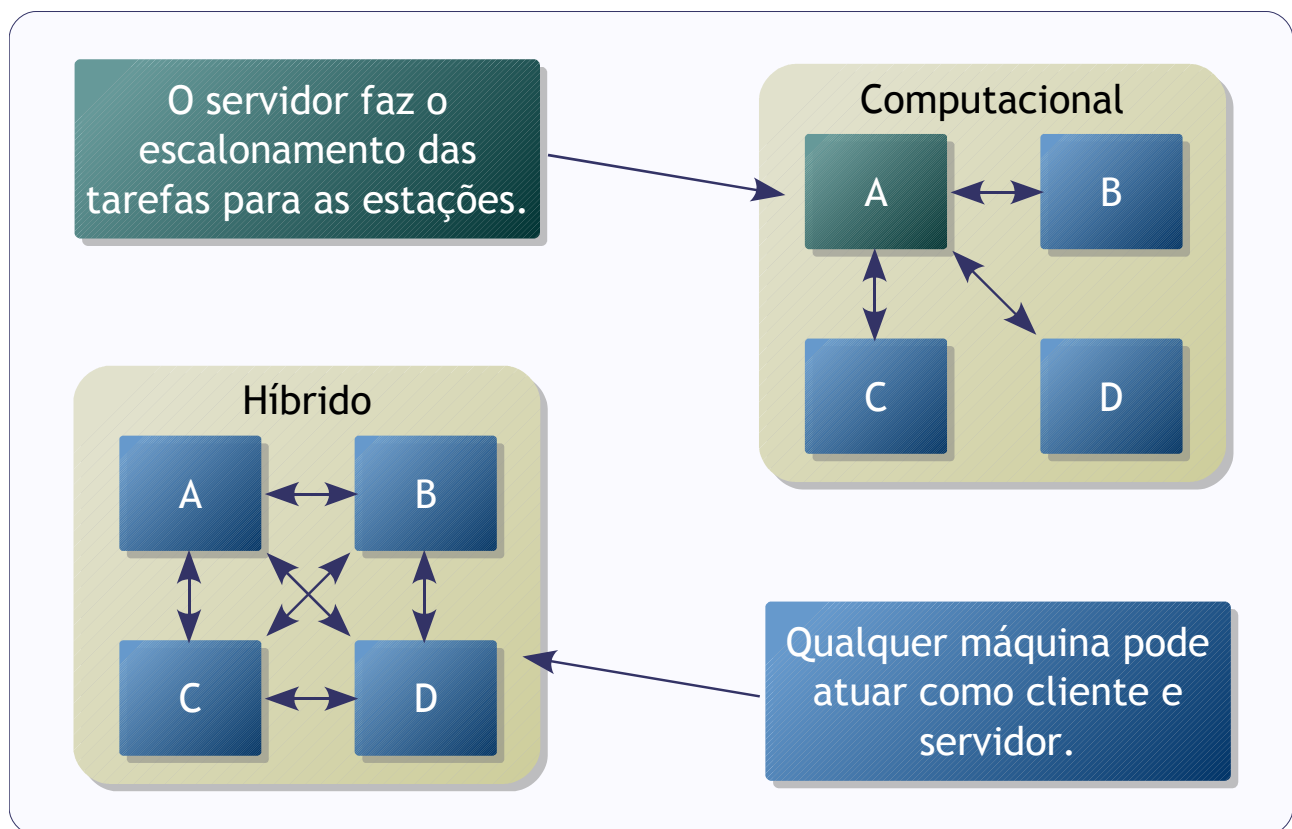
Saída:

```
10.10.10.1 no ar
10.10.10.2 no ar
10.10.10.3 no ar
10.10.10.4 fora do ar
10.10.10.5 no ar
10.10.10.6 fora do ar
10.10.10.7 no ar
10.10.10.8 no ar
10.10.10.9 no ar
10.10.10.10 no ar
```

É importante observar que, quando o processo morre, todas as suas *threads* terminam.

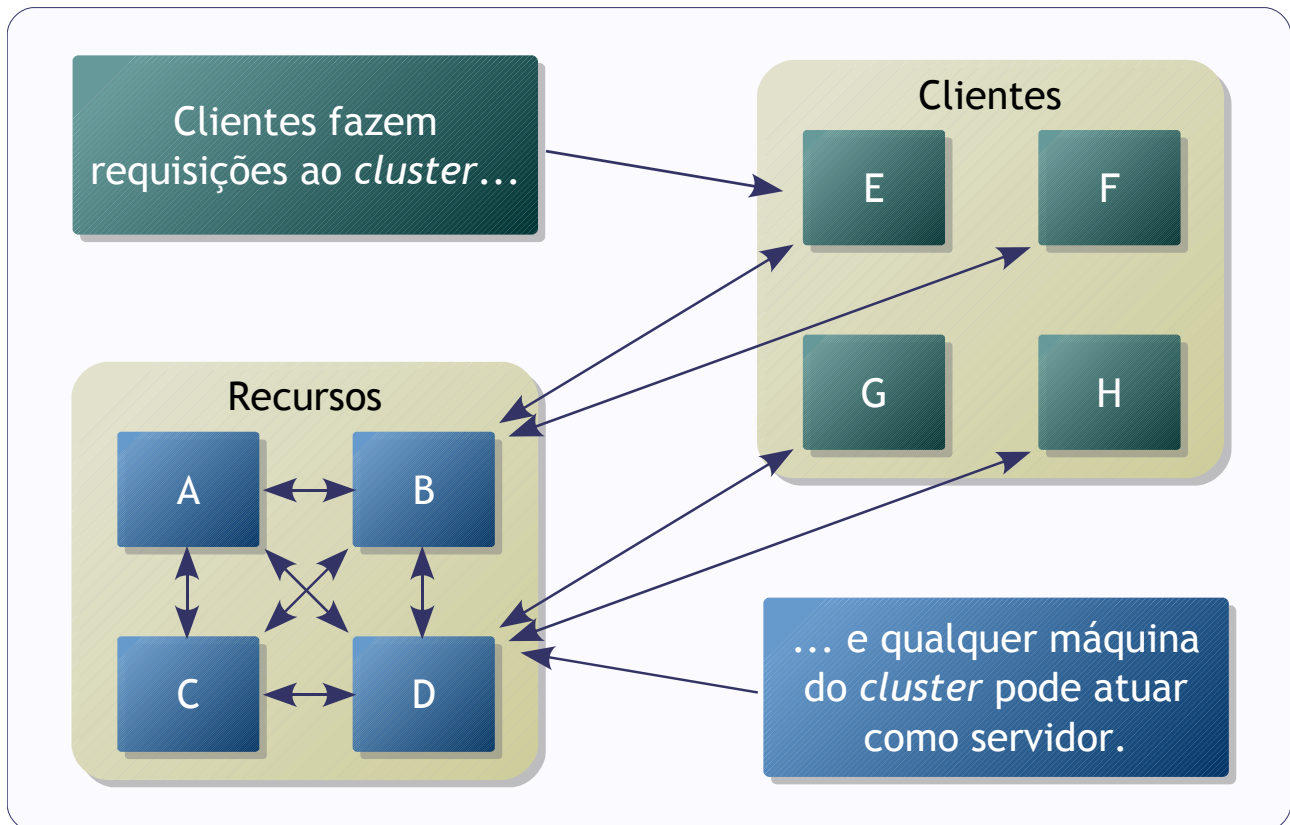
Processamento distribuído

Geralmente a solução para problemas que requerem muita potência computacional é a utilização de máquinas mais poderosas, porém esta solução é limitada em termos de escalabilidade. Uma alternativa é dividir os processos da aplicação entre várias máquinas que se comunicam através de uma rede, formando um *cluster* ou um *grid*.

A diferença básica entre *cluster* e *grid* é que o primeiro tem como premissa de projeto ser um ambiente controlado, homogêneo e previsível, enquanto o segundo é geralmente heterogêneo, não controlado e imprevisível. Um *cluster* é um ambiente planejado especificamente para processamento distribuído, com máquinas dedicadas em um lugar adequado. Um *grid* se caracteriza pelo uso de estações de trabalho que podem estar em qualquer lugar.

Os modelos mais comuns de *cluster*:

- computacional.
- de recursos.
- de aplicação ou híbrido.

O modelo computacional tem como objetivo usar processadores e memória dos

equipamentos envolvidos para obter mais potência computacional. A implementação geralmente utiliza um sistema escalonador de filas (*metascheduler*), que realiza o agendamento das tarefas a serem processados pelos nós (máquinas que compõem o modelo), com isso a operação tende a ser contínua, com interação reduzida com os usuários. Um exemplo conhecido é o SETI@home[58].

O *cluster* de recursos é usado para armazenar informações em um grupo de computadores, tanto para obter mais performance de recuperação de dados quanto para expandir a capacidade de armazenamento. Este modelo pode ser usado para prover infra-estrutura para aplicações ou para atender requisições feitas de forma interativa por usuários. Entre os serviços que podem operar desta forma estão os Sistemas Gerenciadores de Banco de Dados (SGBD), como o MySQL Cluster[59].

O modelo híbrido é basicamente uma aplicação projetada especificamente para funcionar em várias máquinas ao mesmo tempo. Ao invés de prover recursos diretamente, a aplicação utiliza os equipamentos para suportar suas próprias funcionalidades. Com isso, a infra-estrutura é utilizada de forma quase transparente pelos usuários que usam a aplicação interativamente. Todos os nós rodam o aplicativo e podem operar como servidores e clientes. O exemplo mais comum de arquitetura híbrida são os sistemas de

58 Página do projeto em: http://setiathome.berkeley.edu/.
59 Endereço na internet: http://www.mysql.com/products/database/cluster/.

compartilhamento de arquivos (*file sharing*) que usam comunicação *Peer To Peer* (P2P).

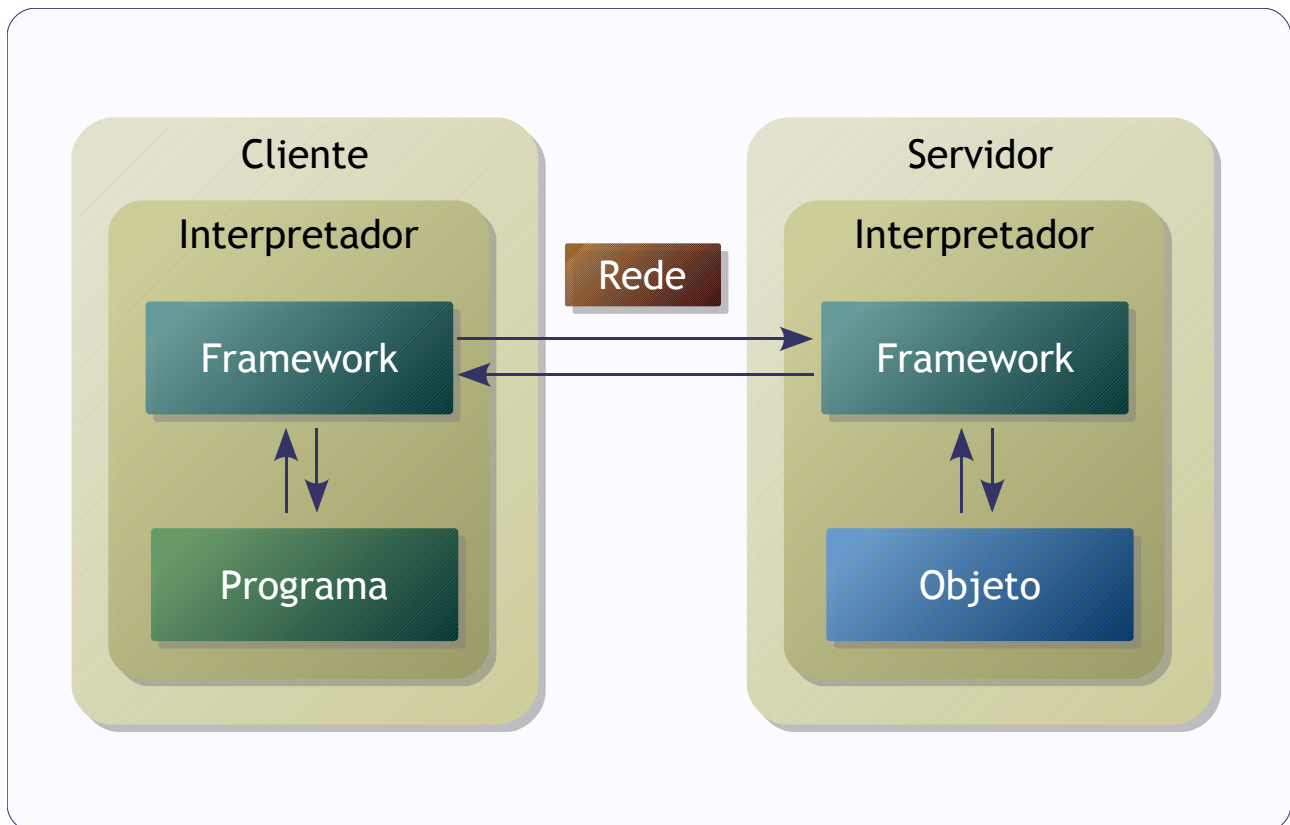
Independente do modelo utilizado. sistemas distribuídos devem atender a quatro requisitos básicos:

- Comunicação: as máquinas envolvidas devem se comunicar de forma permitir a troca de informações entre elas.
- Metadados: os dados sobre o processamento precisam se mantidos de forma adequada.
- Controle: os processos devem ser gerenciados e monitorados.
- Segurança: o sigilo, integridade e disponibilidade devem estar protegidos.

Existem diversas tecnologias voltadas para o desenvolvimento de aplicações distribuídas, tais como: XML-RPC[60], Web Services, objetos distribuídos, MPI e outras.

60 Especificação em http://www.xmlrpc.com/.

Objetos distribuídos

A premissa básica da tecnologia de objetos distribuídos é tornar objetos disponíveis para que seus métodos possam ser evocados remotamente a partir de outras máquinas ou mesmo por outros processos na mesma máquina, usando a pilha de protocolos de rede TCP/IP para isso.

Existem diversas soluções para estes casos, porém utilizar objetos distribuídos oferece várias vantagens em relação a outras soluções que implementam funcionalidades semelhantes, tal como o protocolo XML-RPC:

- Simplicidade para implementação.
- Oculta as camadas de comunicação.
- Suporte a estruturas de dados nativas (contanto que sejam serializáveis).
- Boa performance.
- Maturidade da solução.

PYthon Remote Objects (PYRO[61]) é um *framework* para aplicações distribuídas que permite publicar objetos via TCP/IP. Na máquina servidora, o PYRO publica o objeto, cuidando de detalhes como: protocolo, controle de sessão, autenticação, controle de concorrência e

61 Documentação e fontes disponíveis em: http://pyro.sourceforge.net/.

outros.

Exemplo de servidor:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import Pyro.core

class Dist(Pyro.core.ObjBase):

    def calc(self, n):

        return n**n

if __name__ == '__main__':

    # Inicia a thread do servidor
    Pyro.core.initServer()

    # Cria o servidor
    daemon = Pyro.core.Daemon()

    # Publica o objeto
    uri = daemon.connect(Dist(),'dist')

    # Coloca o servidor em estado operacional
    daemon.requestLoop()
```

Classe de objetos distribuídos.

Na máquina cliente, o programa usa o PYRO para evocar rotinas do servidor e recebe os resultados, da mesma forma que um método de um objeto local.

Exemplo de cliente:

```
# -*- coding: utf-8 -*-

import Pyro.core

# Cria um objeto local para acessar o objeto remoto
proxy = Pyro.core.getProxyForURI('PYROLOC://127.0.0.1/dist')

# Evoca um método do objeto remoto
print proxy.calc(1000)
```

Os métodos publicados através do PYRO não podem ser identificados por introspecção pelo cliente.

Embora o PYRO resolva problemas de concorrência de comunicação com os clientes que estão acessando o mesmo servidor (cada conexão roda em uma *thread* separada), fica por conta do desenvolvedor (ou de outros *frameworks* que a aplicação utilize) resolver questões de concorrência por outros recursos, como arquivos ou conexões de banco de dados[62], por exemplo. É possível autenticar as conexões através da criação de objetos da classe *Validator*, que podem verificar credenciais, endereços IP e outros itens.

62 Problemas de concorrência de conexões de banco de dados podem ser tratados de forma transparente com a utilização de ORMs que implementam esta funcionalidade ou pelo pacote DBUtils (http://www.webwareforpython.org/DBUtils), que faz parte do projeto Webware for Python (http://www.webwareforpython.org/).

Performance

O Python provê algumas ferramentas para avaliar performance e localizar gargalos na aplicação. Entre estas ferramentas estão os módulos *cProfile* e *timeit*.

O módulo *cProfile*[63] faz uma análise detalhada de performance, resultado das chamadas de função, retornos de função e exceções.

Exemplo:

```
# -*- coding: latin1 -*-

import cProfile

def rgb1():
    """
    Função usando range()
    """
    rgbs = []
    for r in range(256):
        for g in range(256):
            for b in range(256):
                rgbs.append('#%02x%02x%02x' % (r, g, b))
    return rgbs

def rgb2():
    """
    Função usando xrange()
    """
    rgbs = []
    for r in xrange(256):
        for g in xrange(256):
            for b in xrange(256):
                rgbs.append('#%02x%02x%02x' % (r, g, b))
    return rgbs

def rgb3():
    """
    Gerador usando xrange()
    """
    for r in xrange(256):
        for g in xrange(256):
            for b in xrange(256):
                yield '#%02x%02x%02x' % (r, g, b)

def rgb4():
```

63 O módulo *cProfile* (disponível no Python 2.5 em diante) é uma versão otimizada do módulo *profile*, que tem a mesma funcionalidade.

```
    """
    Função usando uma lista várias vezes
    """
    rgbs = []
    ints = range(256)
    for r in ints:
        for g in ints:
            for b in ints:
                rgbs.append('#%02x%02x%02x' % (r, g, b))
    return rgbs

def rgb5():
    """
    Gerador usando apenas uma lista
    """
    for i in range(256 ** 3):
        yield '#%06x' % i

def rgb6():
    """
    Gerador usando xrange() uma vez
    """
    for i in xrange(256 ** 3):
        yield '#%06x' % i

# Benchmarks
print 'rgb1:'
cProfile.run('rgb1()')

print 'rgb2:'
cProfile.run('rgb2()')

print 'rgb3:'
cProfile.run('list(rgb3())')

print 'rgb4:'
cProfile.run('rgb4()')

print 'rgb5:'
cProfile.run('list(rgb5())')

print 'rgb6:'
cProfile.run('list(rgb6())')
```

Saída:

```
rgb1:
         16843012 function calls in 54.197 CPU seconds

   Ordered by: standard name
```

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.633    0.633   54.197   54.197 <string>:1(<module>)
        1   49.203   49.203   53.564   53.564 rgbs.py:5(rgb1)
 16777216    4.176    0.000    4.176    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
    65793    0.186    0.000    0.186    0.000 {range}


rgb2:
         16777219 function calls in 53.640 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.624    0.624   53.640   53.640 <string>:1(<module>)
        1   49.010   49.010   53.016   53.016 rgbs.py:16(rgb2)
 16777216    4.006    0.000    4.006    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


rgb3:
         16777219 function calls in 52.317 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.251    6.251   52.317   52.317 <string>:1(<module>)
 16777217   46.066    0.000   46.066    0.000 rgbs.py:27(rgb3)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


rgb4:
         16777220 function calls in 53.618 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.624    0.624   53.618   53.618 <string>:1(<module>)
        1   48.952   48.952   52.994   52.994 rgbs.py:36(rgb4)
 16777216    4.042    0.000    4.042    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {range}


rgb5:
         16777220 function calls in 32.209 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.110    6.110   32.209   32.209 <string>:1(<module>)
```

```
 16777217   25.636    0.000   26.099    0.000 rgbs.py:48(rgb5)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.463    0.463    0.463    0.463 {range}


rgb6:
         16777219 function calls in 30.431 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.066    6.066   30.431   30.431 <string>:1(<module>)
 16777217   24.365    0.000   24.365    0.000 rgbs.py:55(rgb6)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

O relatório do *cProfile* mostra no inicio as duas informações mais importantes: o tempo de CPU consumido em segundos e a quantidade de chamadas de função. As outras linhas mostram os detalhes por função, incluindo o tempo total e por chamada.

As cinco rotinas do exemplo têm a mesma funcionalidade: geram uma escala de cores RGB. Porém, o tempo de execução é diferente.

Analisando os resultados:

Rotina	Tipo	Tempo	Laços	x/range()
rgb1()	Função	54.197	3	range()
rgb2()	Função	53.640	3	xrange()
rgb3()	Gerador	52.317	3	xrange()
rgb4()	Função	53.618	3	range()
rgb5()	Gerador	32.209	1	range()
rgb6()	Gerador	30.431	1	xrange()

Fatores observados que pesaram no desempenho:

- A complexidade do algoritmo.
- Geradores apresentaram melhores resultados do as funções tradicionais.
- O gerador *xrange()* consomem menos recursos do que a função *range()*.

O gerador *rgb6()*, que usa apenas um laço e *xrange()*, é bem mais eficiente que as outras rotinas.

Outro exemplo:

```
# -*- coding: latin1 -*-

import cProfile

def fib1(n):
    """
    Fibonacci calculado de forma recursiva.
    """
    if n > 1:
        return fib1(n - 1) + fib1(n - 2)
    else:
        return 1

def fib2(n):
    """
    Fibonacci calculado por um loop.
    """
    if n > 1:

        # O dicionário guarda os resultados
        fibs = {0:1, 1:1}
        for i in xrange(2, n + 1):
            fibs[i] = fibs[i - 1] + fibs[i - 2]
        return fibs[n]
    else:
        return 1

print 'fib1'
cProfile.run('[fib1(x) for x in xrange(1, 31)]')
print 'fib2'
cProfile.run('[fib2(x) for x in xrange(1, 31)]')
```

Saída:

```
fib1
         7049124 function calls (32 primitive calls) in 21.844 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   21.844   21.844 <string>:1(<module>)
7049122/30   21.844    0.000   21.844    0.728 fibs.py:4(fib1)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


fib2
         32 function calls in 0.001 CPU seconds

   Ordered by: standard name
```

```
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    0.001    0.001 <string>:1(<module>)
      30    0.001    0.000    0.001    0.000 fibs.py:13(fib2)
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

A performance do cálculo da série de Fibonacci usando um laço que preenche um dicionário é muito mais eficiente do que a versão usando recursão, que faz muitas chamadas de função.

O módulo *timeit* serve para fazer *benchmark* de pequenos trechos de código[64]. O módulo foi projetado para evitar as falhas mais comuns que afetam programas usados para fazer *benchmarks*.

Exemplo:

```
import timeit

# Lista dos quadrados de 1 a 1000
cod = '''s = []
for i in xrange(1, 1001):
    s.append(i ** 2)
'''
print timeit.Timer(cod).timeit()

# Com Generator Expression
cod = 'list(x ** 2 for x in xrange(1, 1001))'
print timeit.Timer(cod).timeit()

# Com List Comprehesion
cod = '[x ** 2 for x in xrange(1, 1001)]'
print timeit.Timer(cod).timeit()
```

O código é passado como texto.

Saída:

```
559.372637987
406.465490103
326.330293894
```

O *List Comprehension* é mais eficiente do que o laço tradicional.

Outra forma de melhorar a performance de uma aplicação é usando o *Psyco*, que é uma

64 O módulo *cProfile* não é apropriado para avaliar pequenos trechos de código. O módulo *timeit* é mais adequado pois executa o código várias vezes durante a avaliação.

espécie de *Just In Compiler* (JIT). Durante a execução, ele tenta otimizar o código da aplicação e, por isso, o módulo deve ser importado antes do código a ser otimizado (o inicio do módulo principal da aplicação é um lugar adequado).

Exemplo (com o último trecho de código avaliado no exemplo anterior):

```
import psyco

# Tente otimizar tudo
psyco.full()

import timeit

# Lista dos quadrados de 1 a 1000
cod = '[x ** 2 for x in xrange(1, 1001)]'
print timeit.Timer(cod).timeit()
```

Saída:

```
127.678481102
```

O código foi executado mais de duas vezes mais rápido do que antes. Para isso, foi necessário apenas acrescentar duas linhas de código.

Porém, o *Psyco* deve ser usado com alguns cuidados, pois em alguns casos ele pode não conseguir otimizar ou até piorar a performance[65]. As funções *map()* e *filter()* devem ser evitadas e módulos escritos em C, como o *re* (expressões regulares) devem ser marcados com a função *cannotcompile()* para que o *Psyco* os ignore. O módulo fornece formas de otimizar apenas determinadas partes do código da aplicação, tal como a função *profile()*, que só otimiza as partes mais pesadas do aplicativo, e uma função *log()* que analisa a aplicação, para contornar estas situações.

Algumas dicas sobre otimização:

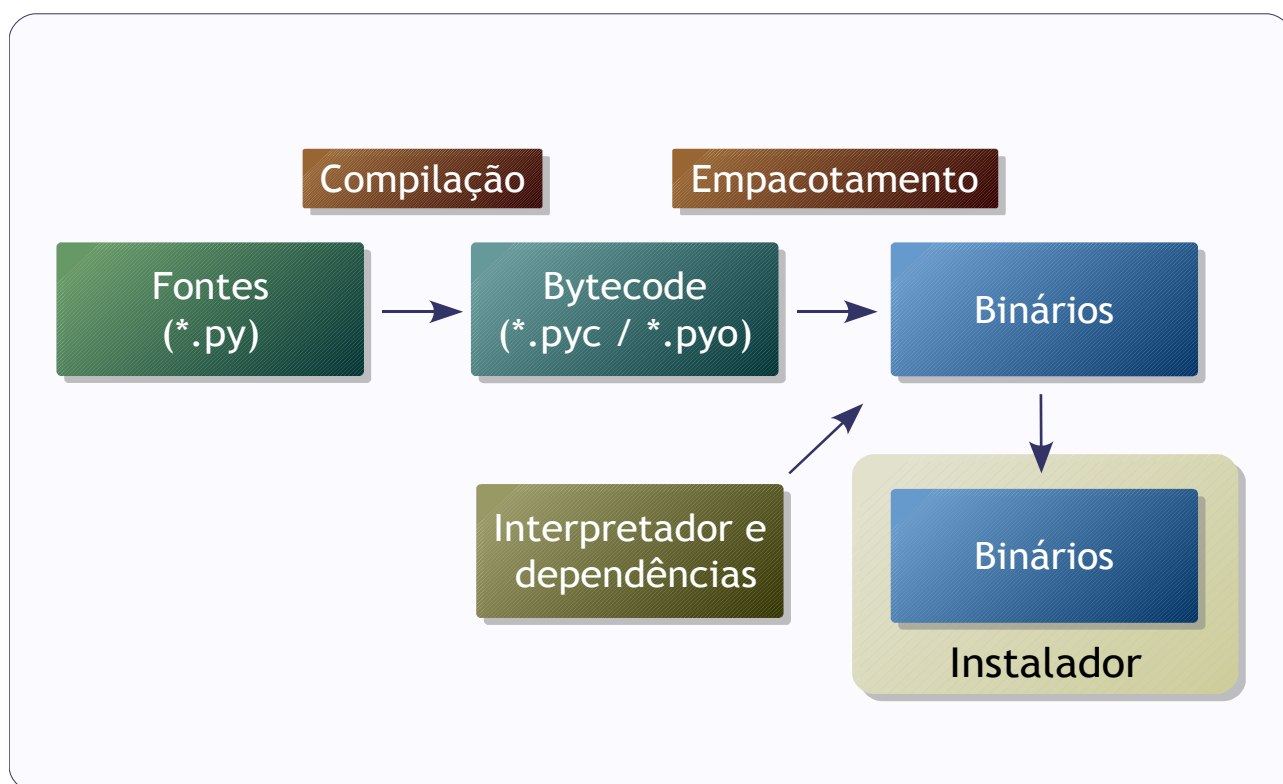
- Mantenha o código simples.
- Otimize apenas o código aonde a performance da aplicação é realmente crítica.
- Use ferramentas para identificar os gargalos no código.
- Evite funções recursivas.
- Use os recursos nativos da linguagem. As listas e dicionários do Python são muito otimizados.

65 Existem algumas funções em que o Psyco tem o efeito de reduzir a velocidade do programa, pois o próprio *Psyco* consome CPU também. Além disso, o *Psyco* também faz com que a aplicação consuma mais memória.

- Use *List Comprehensions* ao invés de laços para processar listas usando expressões simples.
- Evite funções dentro de laços. Funções podem receber e devolver listas.
- Use geradores ao invés de funções para grandes seqüências de dados.

Empacotamento e distribuição de aplicações

Geralmente é bem mais simples distribuir aplicações na forma de binário, em que basta rodar um executável para iniciar a aplicação, que instalar todas as dependências necessárias em cada máquina aonde se deseja executar a aplicação.

Existem vários softwares que permitem gerar executáveis a partir de um programa feito em Python, como o Py2exe[66] e cx_Freeze[67].

O Py2exe só funciona na plataforma Windows, porém possui muitos recursos, podendo gerar executáveis com interface de texto, gráficos, serviços (programas que rodam sem intervenção do usuário, de forma semelhante aos *daemons* nos sistemas UNIX) e servidores COM (arquitetura de componentes da Microsoft).

O cx_Freeze é portável, podendo rodar em ambientes UNIX, porém é bem menos versátil que o Py2exe.

Para usar o Py2exe, é preciso criar um *script*, que normalmente se chama "setup.py", que

66 Documentação, fontes e binários de instalação podem ser encontrados em: http://www.py2exe.org/.
67 Documentação, fontes e binários de instalação para várias plataformas podem ser encontrados em: http://starship.python.net/crew/atuining/cx_Freeze/.

diz ao Py2exe o que é necessário para gerar o executável.

Exemplo de “setup.py”:

```
# -*- coding: latin1 -*-
"""
Exemplo de uso do py2exe
"""
from distutils.core import setup
import py2exe

setup(name = 'SIM - Sistema Interativo de Música',
    service = ['simservice'],
    console = ['sim.py', 'simimport.py'],
    windows = ['simgtk.py'],
    options = {'py2exe': {
        'optimize': 2,
        'includes': ['atk', 'gobject', 'gtk','gtk.glade',
        'pango', 'cairo', 'pangocairo']
    }},
    data_files=[('',['janela.glade', 'sim.ico'])],
    description = 'Primeira Versão...',
    version = '1.0')
```

No exemplo, temos um sistema que é composto por dois utilitários de linha comando, um aplicativo com interface gráfica e um serviço. O aplicativo com GUI depende do GTK+ para funcionar e foi desenvolvido usando Glade.

Entre os parâmetros do Py2exe, os mais usuais são:

- *name*: nome da aplicação.
- *service*: lista de serviços.
- *console*: lista de programas com interface de texto.
- *windows*: lista de programas com interface gráfica.
- *options['py2exe']*: dicionário com opções que alteram o comportamento do Py2exe:
- *optimize*: 0 (otimização desativada, *bytecode* padrão), 1 (otimização ativada, equivale ao parâmetro “-O” do interpretador) ou 2 (otimização com remoção de *Doc Strings* ativada, equivale ao parâmetro “-OO” do interpretador)
- *includes*: lista de módulos que serão incluídos como dependências. Geralmente, o *Py2exe* detecta as dependências sem necessidade de usar esta opção.
- *data_files*: outros arquivos que fazem parte da aplicação, tais como imagens, ícones e arquivos de configuração.
- *description*: comentário.
- *version*: versão da aplicação, como *string*.

Para gerar o executável, o comando é:

```
python setup.py py2exe
```

O Py2exe criará duas pastas:

- *build*: arquivos temporários.
- *dist*: arquivos para distribuição.

Entre os arquivos para distribuição, "w9xpopen.exe" é necessário apenas para as versões antigas do Windows (95 e 98) e pode ser removido sem problemas em versões mais recentes.

Pela linha de comando também é possível passar algumas opções interessantes, como o parâmetro "-b1", para gerar menos arquivos para a distribuição.

O cx_Freeze é um utilitário de linha de comando.

```
FreezePython -OO -c sim.py
```

A opção "-c" faz com que o *bytecode* seja comprimido no formato *zip*.

```
FreezePython -OO --include-modules=atk,cairo,pango,pangocairo simgtk.py
```

A opção "--include-modules", permite passar uma lista de módulos que serão incluídos na distribuição.

Tanto o Py2exe quanto o cx_Freeze não são compiladores. O que eles fazem é empacotar os *bytecodes* da aplicação, suas dependências e o interpretador em si em (pelo menos) um arquivo executável (e arquivos auxiliares) que não dependem do ambiente aonde foram gerados. Com isso, a distribuição do aplicativo se torna bem mais simples. Entretanto, não há ganho de performance em gerar executáveis, tirando na carga da aplicação para a memória em alguns casos.

Eles também não geram programas de instalação. Para isso, é necessário o uso de um software específico. Os instaladores são gerados por aplicativos que se encarregam de automatizar tarefas comuns do processo de instalação. São exemplos de softwares dessa categoria: Inno Setup[68] e NSIS[69].

68 Documentação e binários de instalação disponíveis em: http://www.jrsoftware.org/isinfo.php.
69 Endereço do projeto: http://nsis.sourceforge.net/Main_Page.

Exercícios VI

1. Implementar um módulo com uma função *tribonacci(n)* que retorne uma lista de *n* números de Tribonacci, aonde *n* é o parâmetro da função. Faça testes da função caso o módulo seja executado como principal.

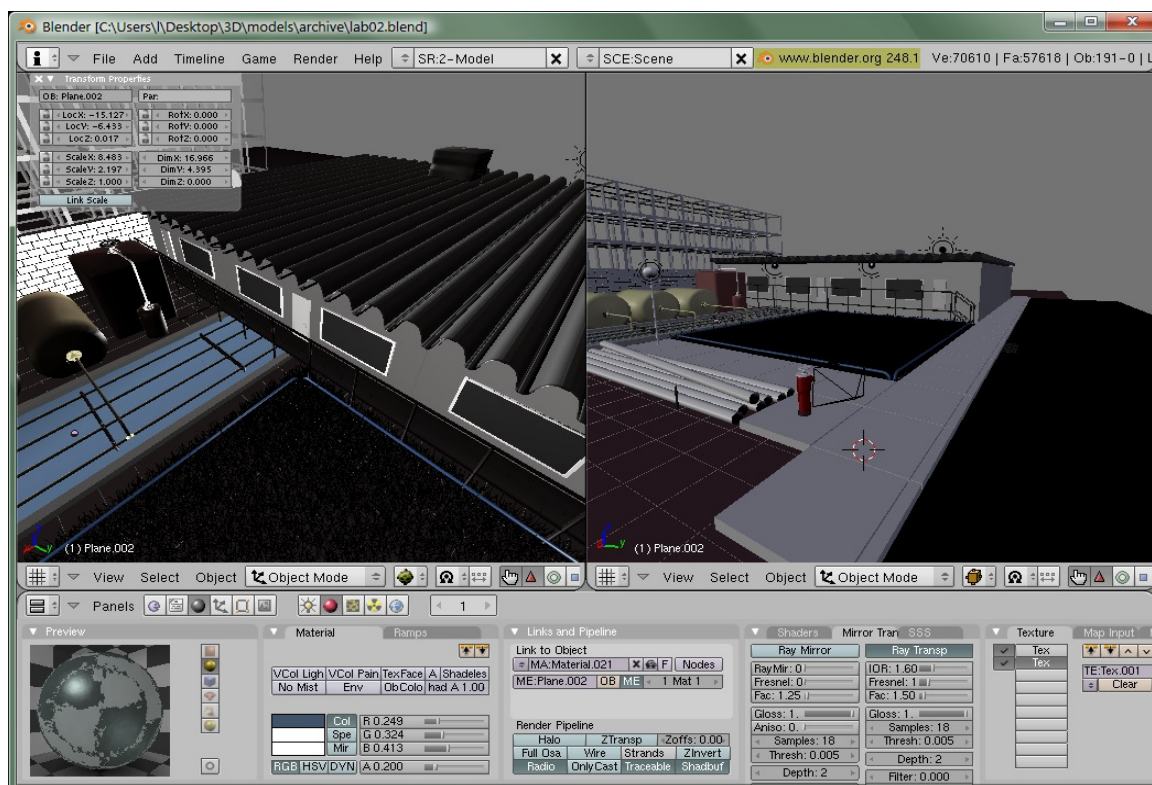
2. Implementar:

- um servidor que publique um objeto distribuído e este evoque a função *tribonacci*.
- um cliente que use o objeto distribuído para calcular a seqüência de Tribonacci.

Apêndices

- Integração com Blender.
- Integração com BrOffice.org.
- Integração com Linguagem C.
- Integração com .NET.
- Respostas dos exercícios.

Integração com Blender

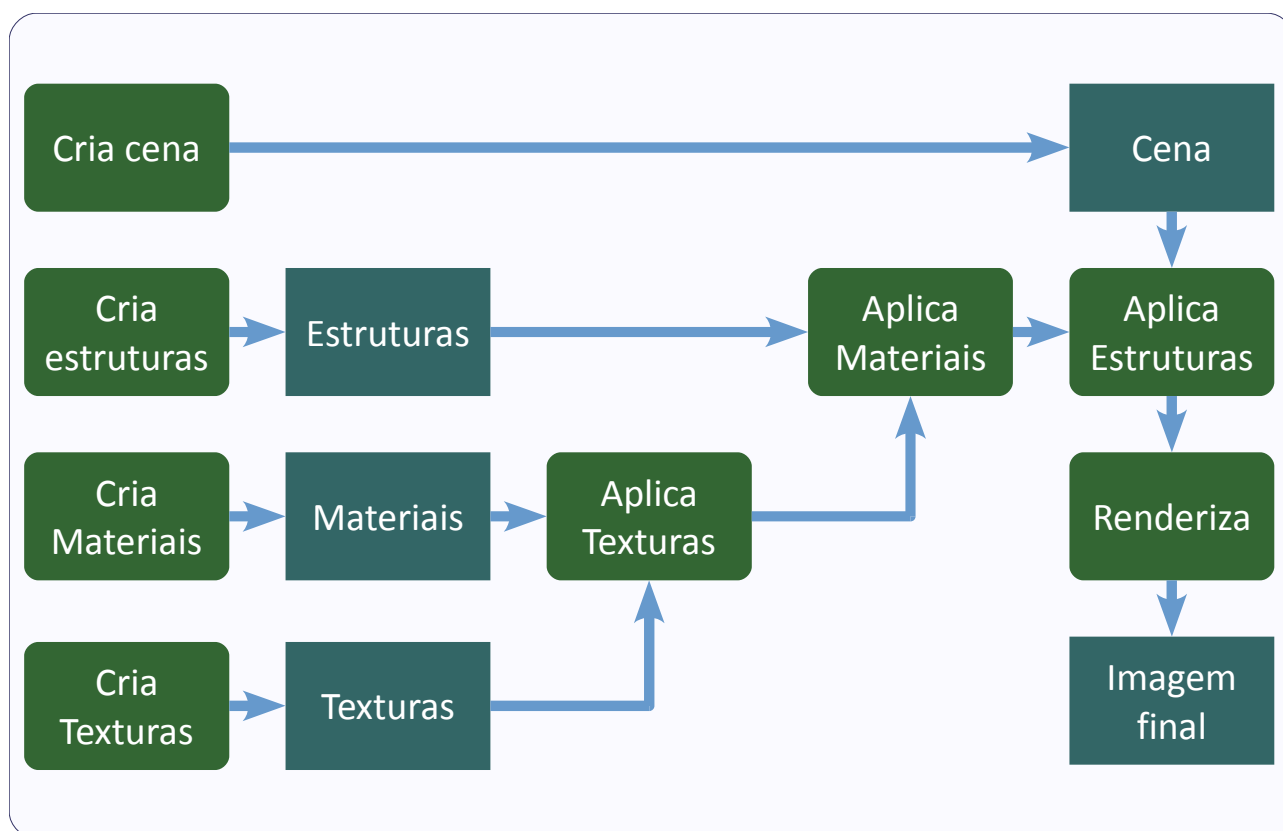
Python pode ser usado como linguagem *script* em várias aplicações para automatizar tarefas e/ou adicionar novas funcionalidades. Entre elas, está o Blender[70], que é um software de modelagem 3D de código aberto, que pode gerar animações e também pode ser usado como *Game Engine* (infraestrutura especializada para criação de jogos para computador).

No Blender, uma cena é composta por objetos, que precisam ser fixados em posições e conectados a cena. Se o objeto não estiver conectado a cena, ele é eliminado ao fim do processamento. Normalmente, uma cena contém pelo menos uma fonte de luz, uma câmera e *meshes* (malhas que representam os objetos 3D), que podem usar vários materiais e estes podem ter várias texturas. Texturas são imagens bidimensionais (procedurais ou *raster*) que podem ser usadas nos materiais aplicados as superfícies dos objetos, alterando várias propriedades, tais como reflexão, translucência, coloração e enrugamento (*bump*) da superfície.

Uma malha é um conjunto de pontos (vértices), interligados por linhas (arestas) que formam superfícies (faces) de um objeto.

70 Documentação, fontes e binários podem ser encontrados em: http://www.blender.org/.

Com Python é possível acessar todas essas estruturas do Blender através de módulos:

- *Blender*: permite abrir arquivos, salvar e outras funções correlatas.
- *Object*: operações com objetos 3D.
- *Materials*: manipulação de materiais.
- *Textures*: manipulação de texturas.
- *World*: manipulação do ambiente da cena.

Fluxo para a criação de um cenário 3D no Blender:

Exemplo de código para a criação de uma cena:

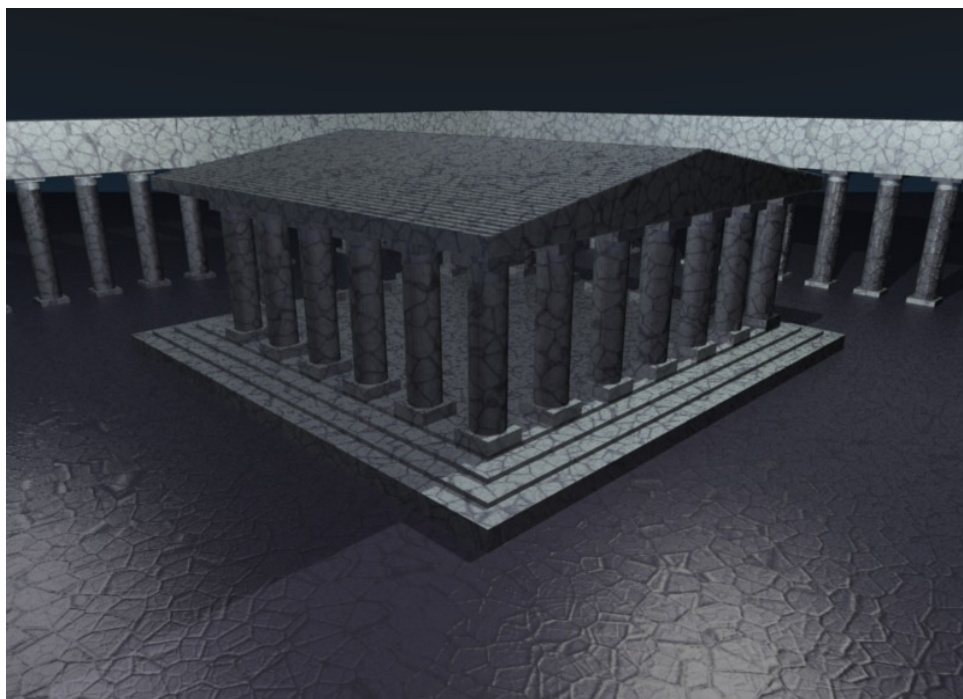
```
import math
import Blender

# Pega a cena atual
cena = Blender.Scene.GetCurrent()

# Elementos da cena "default"
camera = Blender.Object.Get()[0]
cubo = Blender.Object.Get()[1]
lamp = Blender.Object.Get()[2]
```

```
# Move a camera
camera.setLocation(8., -8., 4.)
camera.setEuler(math.radians(70), 0.,
   math.radians(45))
# Muda a lente
camera.data.lens = 25

# Remove da cena o objeto "default"
cena.objects.unlink(cubo)

# Altera a intensidade da luz
lamp.data.energy = 0.8
# E a cor
lamp.data.col = (1., 1., 0.9)

# Cria outra fonte de luz
lamp1 = Blender.Lamp.New('Lamp')
lamp1.energy = 0.7
lamp1.col = (0.9, 1., 1.)
_lamp1 = Blender.Object.New('Lamp')

# Muda o lugar da fonte (default = 0.0, 0.0, 0.0)
_lamp1.setLocation(6., -6., 6.)

# "Prende" a fonte de luz na cena
_lamp1.link(lamp1)
cena.objects.link(_lamp1)

# Cria outra fonte de luz
lamp2 = Blender.Lamp.New('Lamp')
lamp2.energy = 0.7
lamp1.col = (1., 0.9, 1.)
_lamp2 = Blender.Object.New('Lamp')
_lamp2.setLocation(-6., -6., 6.)
_lamp2.link(lamp2)
cena.objects.link(_lamp2)

# Cria um material
material1 = Blender.Material.New('newMat1')
material1.rgbCol = [.25, .25, .32]
material1.setAlpha(1.)

# Cria uma textura
textura1 = Blender.Texture.Get()[0]
textura1.setType('Clouds')
textura1.noiseType = 'soft'
textura1.noiseBasis = Blender.Texture.Noise['VORONOICRACKLE']

# Coloca no material
material1.setTexture(0, textura1)
mtex1 = material1.getTextures()[0]
mtex1.col = (.18, .18, .22)
```

```
mtex1.mtNor = 1
mtex1.neg = True
mtex1.texco = Blender.Texture.TexCo['GLOB']
material1.mode += Blender.Material.Modes['RAYMIRROR']
material1.rayMirr = 0.3
material1.glossMir = 0.6

# Cria o piso
mesh = Blender.Mesh.Primitives.Plane(40.)
piso = cena.objects.new(mesh,'Mesh')
# Rotaciona o piso
piso.setEuler(0., 0., math.radians(45))

# Coloca o material no piso
piso.setMaterials([material1])
piso.colbits = 1

# Cria outro material
material2 = Blender.Material.New('newMat2')
material2.rgbCol = [.68, .76, .77]
material2.setAlpha(1.)
material2.mode += Blender.Material.Modes['RAYMIRROR']
material2.rayMirr = 0.2
material2.glossMir = 0.8

# Coloca textura no outro material
material2.setTexture(0, textura1)
mtex2 = material2.getTextures()[0]
mtex2.col = (.22, .22, .32)
mtex2.mtNor = 1
mtex2.neg = True
mtex2.texco = Blender.Texture.TexCo['GLOB']

# Cria objetos na cena
def objeto(local, tam, mat, prim=Blender.Mesh.Primitives.Cube):

    mesh = prim()
    obj = cena.objects.new(mesh, 'Mesh')
    obj.setLocation(*local)
    obj.size = tam
    obj.setMaterials(mat)
    obj.colbits = 1
    return obj

mat = [material2]

# Cria colunas no fundo
for i in xrange(16):

    # Topo da primeira fileira
    local = (i - 8., 8., 2.5)
    tam = (.25, .25, .1)
```

```
    objeto(local, tam, mat)

    # Base da primeira fileira
    local = (i - 8., 8., 0.)
    objeto(local, tam, mat)

    # Corpo da primeira fileira
    prim = Blender.Mesh.Primitives.Cylinder
    tam = .2, .2, .25

    for k in xrange(10):
        local = (i - 8., 8., .25 * k)
        objeto(local, tam, mat, prim)

    # Topo da segunda fileira
    local = (-8., i - 8., 2.5)
    tam = (.25, .25, .1)
    objeto(local, tam, mat)

    # Base da segunda fileira
    local = (-8., i - 8., 0.)
    objeto(local, tam, mat)

    # Corpo da segunda fileira
    tam = .2, .2, .25

    for k in xrange(10):
        local = (-8., i - 8., .25 * k)
        objeto(local, tam, mat, prim)

    # Aqueduto
    local = (-8., i - 8., 3.)
    tam = (.5, .5, .5)
    objeto(local, tam, mat)

    local = (i - 8., 8., 3.)
    objeto(local, tam, mat)

# Cria colunas em cima do piso
for i in (-3, 3):
    for j in range(-2, 4):

        # Topo das fileiras X
        local = (i, j, 2.5)
        tam = (.25, .25, .1)
        objeto(local, tam, mat)

        # Topo das fileiras Y
        local = (j, i, 2.5)
        objeto(local, tam, mat)

        # Base das fileiras X
```

```
        local = (i, j, .5)
        objeto(local, tam, mat)

        # Base das fileiras Y
        local = (j, i, .5)
        objeto(local, tam, mat)

        tam = (.2, .2, .25)
        # Corpo das fileiras X
        for k in xrange(10):
            local = (i, j, .25 * k)
            objeto(local, tam, mat, prim)

        # Corpo das fileiras Y
        for k in xrange(10):
            local = (j, i, .25 * k)
            objeto(local, tam, mat, prim)

# Cria escada
for i in xrange(4):

    local = (0., 0., i / 16.)
    tam = (2. + (8. - i) / 4.,2. + (8. - i) / 4., .25)
    objeto(local, tam, mat)

# Cria teto
for i in xrange(35):

    local = (0., 0., 2.7 + i / 60.)
    tam = (3.5 , 3.5 * ( 1. - i / 35.), .1)
    objeto(local, tam, mat)

# Pega o "mundo"
world = Blender.World.Get()[0]
# Modo "blend" no fundo
world.skytype = 1

# Atualiza a cena
cena.update()
```

Saída (renderizada através da câmera padrão):

Para executar código Python dentro do Blender, é só carregar o programa na janela de editor de texto do Blender e usar a opção de execução no menu.

Integração com BrOffice.org

BrOffice.org é um conhecido pacote de automação de escritórios de código aberto, que inclui editor de textos, planilha e outros aplicativos. Além disso, o BrOffice.org também possui:

- Suporte ao Python como linguagem de macro, permitindo a automatização de tarefas e a construção de extensões (*add ons*).
- Um serviço para atender conexões, através de uma API chamada UNO (*Universal Network Objects*), acessível através do Python, entre outras linguagens.

Exemplo de geração de relatório em PDF através do editor de texto (Writer), através da Python UNO Bridge:

```
# -*- coding: latin1 -*-

# Para iniciar o BrOffice.org como servidor:
# swriter.exe -headless
#   "-accept=pipe,name=py;urp;StarOffice.ServiceManager"

import os
import uno
from com.sun.star.beans import PropertyValue

# Dados...
mus = [('Artista', 'Faixa'),
   ('King Crimson', 'Starless'), ('Yes', 'Siberian Khatru'),
   ('Led Zeppellin', 'No Quarter'), ('Genesis', 'Supper\'s Ready')]

rows = len(mus)
cols = len(mus[0])

# Inicio do "Boiler Plate"...

# Contexto de componente local
loc = uno.getComponentContext()

# Para resolver URLs
res = loc.ServiceManager.createInstanceWithContext(
   'com.sun.star.bridge.UnoUrlResolver', loc)

# Contexto para a URL
con = res.resolve('uno:pipe,name=py;urp;StarOffice.ComponentContext')

# Documento corrente
desktop = con.ServiceManager.createInstanceWithContext(
   'com.sun.star.frame.Desktop', con)

# Fim do "Boiler Plate"...
```

```
# Cria um documento novo no Writer
doc = desktop.loadComponentFromURL('private:factory/swriter',
   '_blank', 0, ())

# Cursor de texto
cursor = doc.Text.createTextCursor()

# Muda propriedades
cursor.setPropertyValue('CharFontName', 'Verdana')
cursor.setPropertyValue('CharHeight', 20)
cursor.setPropertyValue('CharWeight', 180)
doc.Text.insertString(cursor, 'Músicas favoritas\n', 0)

# Cria tabela
tab = doc.createInstance('com.sun.star.text.TextTable')
tab.initialize(rows, cols)
doc.Text.insertTextContent(cursor, tab, 0)

# Preenche a tabela
for row in xrange(rows):
   for col in xrange(cols):
      cel = chr(ord('A') + col) + str(row + 1)
      tab.getCellByName(cel).setString(mus[row][col])

# Propriedades para exportar o documento
props = []
p = PropertyValue()
p.Name = 'Overwrite'
p.Value = True # Sobrescreve o documento anterior
props.append(p)

p = PropertyValue()
p.Name = 'FilterName'
p.Value = 'writer_pdf_Export' # Writer para PDF
props.append(p)

# URL de destino
url = uno.systemPathToFileUrl(os.path.abspath('musicas.pdf'))

# Salva o documento como PDF
doc.storeToURL(url, tuple(props))

# Fecha o documento
doc.close(True)
```

Saída (no arquivo PDF):

Músicas favoritas

Artista	Faixa
King Crimson	Starless
Yes	Siberian Khatru
Led Zeppellin	No Quarter
Genesis	Supper's Ready

A API do BrOffice.org é bastante completa e simplifica várias atividades que são lugar comum em programas para ambiente *desktop*.

Integração com Linguagem C

É possível integrar Python e C nos dois sentidos:

- Python => C (Python faz chamadas a um módulo compilado em C).
- C => Python (C evoca o interpretador Python em modo *embedded*).

Python => C

O módulo escrito em C deve utilizar as estruturas do Python (que estão definidas na API de interface) para se comunicar com o interpretador Python.

Exemplo:

```
// Arquivo: mymodule.c

// Python.h define as estruturas do Python em C
#include <Python.h>

// No Python, mesmo os erros sao objetos
static PyObject *MyModuleError;

// Chamando a funcao "system" em C
static PyObject *
mymodule_system(PyObject *self, PyObject *args)
{
  const char *command;
  int sts;

  // "PyArg_ParseTuple" desempacota a tupla de parametros
  // "s" significa que ele deve identificar uma string
  if (!PyArg_ParseTuple(args, "s", &command))
    // retornando NULL gera uma excessao
    // caso falte parametros
    return NULL;

  // chamando "system":
  sts = system(command);

  // "Py_BuildValue" gera objetos que o Python conhece
  // "i" significa inteiro
  return Py_BuildValue("i", sts);
}

// Tabela que o Python consulta para resolver
// os metodos do modulo e pode ser usado
// tambem para gerar a documentacao
// por instrospeccao: dir(), help(),...
static PyMethodDef MyModuleMethods[] = {
```

```
  {"system", mymodule_system, METH_VARARGS,
  "Executa comandos externos."},
  // Fim da tabela:
  {NULL, NULL, 0, NULL}
};

// inicializacao do modulo:
PyMODINIT_FUNC
initmymodule(void)
{
  // O modulo tambem e' um objeto
  PyObject *m;

  // "Py_InitModule" precisa do nome do modulo e da
  // tabela de metodos
  m = Py_InitModule("mymodule", MyModuleMethods);

  // Erros...
  MyModuleError = PyErr_NewException("mymodule.error",
    NULL, NULL);

  // "Py_INCREF" incrementa o numero de referencias do objeto
  Py_INCREF(MyModuleError);

  // "PyModule_AddObject" adiciona um objeto ao modulo
  PyModule_AddObject(m, "error", MyModuleError);
}
```

Ao invés de compilar o módulo manualmente, use o Python para automatizar o processo. Primeiro, crie o *script*:

```
# Arquivo: setup.py

from distutils.core import setup, Extension

mymodule = Extension('mymodule', sources = ['mymodule.c'])
setup(name = 'MyPackage', version = '1.0',
    description = 'My Package',
    ext_modules = [mymodule])
```

E para compilar:

```
python setup.py build
```

O binário compilado será gerado dentro da pasta "build". O módulo pode ser usado como qualquer outro módulo no Python (através de *import*).

C => Python

O inverso também é possível. Um programa escrito em C pode evocar o interpretador Python seguindo três passos:

- Inicializar o interpretador.
- Interagir (que pode ser feito de diversas formas).
- Finalizar o interpretador.

Exemplo:

```
// Arquivo: py_call.c

// Python.h com as definicoes para
// interagir com o interpretador
#include <Python.h>

int main()
{
 // Inicializa interpretador Python
 Py_Initialize();

 // Executando codigo Python
 PyRun_SimpleString("import os\n"
   "for f in os.listdir('.'):\n"
   "  if os.path.isfile(f):\n"
   "    print f, ':', os.path.getsize(f)\n");

 // Finaliza interpretador Python
 Py_Finalize();
 return 0;
}
```

Para compilar, é preciso passar a localização das *headers* e *libraries* do Python para o compilador C:

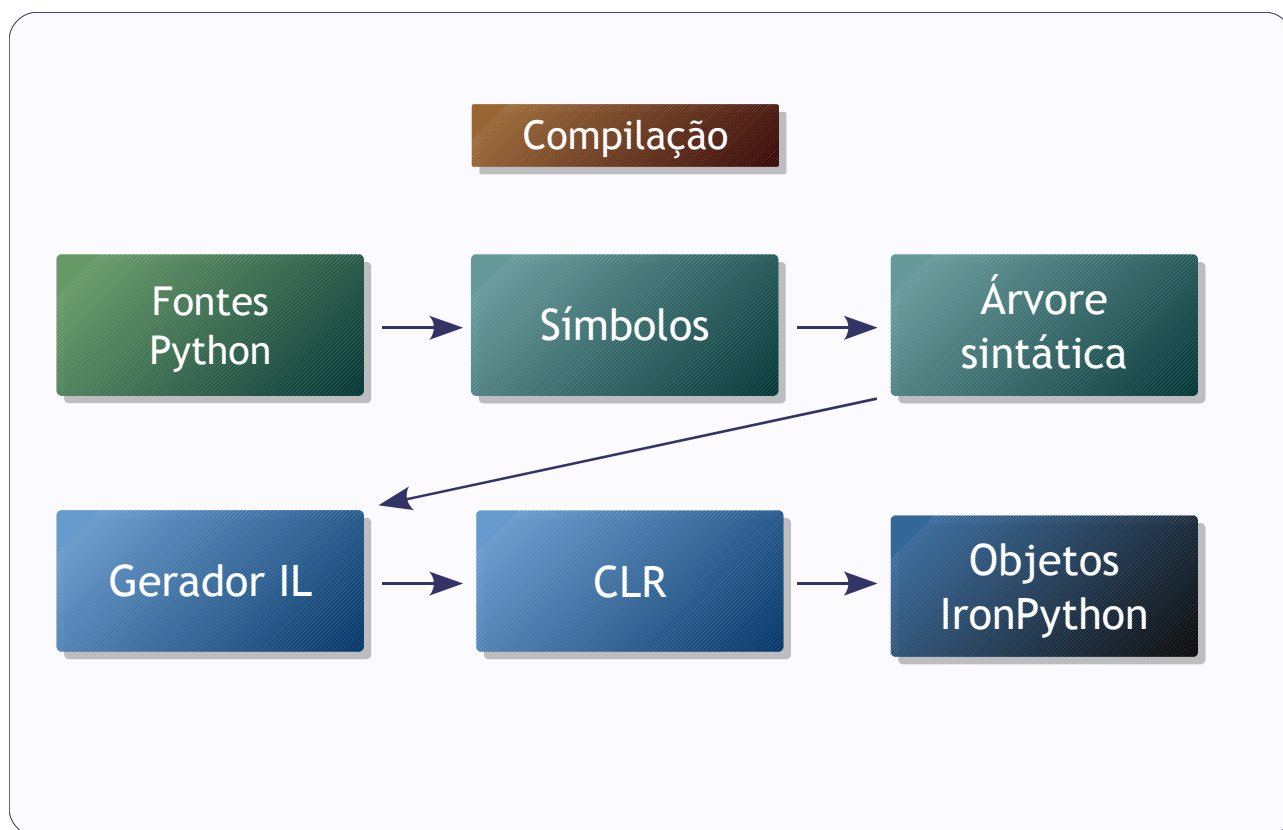
```
gcc -I/usr/include/python2.5 \
 -L/usr/lib/python2.5/config \
 -lpython2.5 -opy_call py_call.c
```

Observações:

- Esta API faz parte do CPython (porte do Python escrito em C).
- Existem ferramentas para automatizar o processo para gerar interfaces para sistemas maiores: SWIG, Boost.Python e SIP.

Integração com .NET

IronPython[71] é a implementação do interpretador Python na linguagem C#. Embora o projeto tenha como objetivo a compatibilidade com CPython, existem algumas diferenças entre elas. A principal vantagem do IronPython em relação ao CPython é a integração com componentes baseados no *framework* .NET.

O .NET é uma infra-estrutura de software criada pela Microsoft para a criação e execução de aplicações. A parte principal do .NET é o *Common Language Runtime* (CLR), que provê uma série recursos aos programas, como gerenciamento de memória para as aplicações. Além disso, há um vasto conjunto de bibliotecas de componentes prontos para uso. As instruções das linguagens de programação são traduzidas para *intermediate language* (IL) reconhecida pelo CLR, permitindo que várias linguagens sejam usadas.

Dentro dos recursos disponíveis no *framework*, existe o *Dynamic Language Runtime* (DLR), que implementa os serviços necessários para linguagens dinâmicas. O IronPython faz uso desses serviços.

71 Fontes, binários, exemplos, documentação e outras informações podem ser encontrados em: http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython.

Para evocar o modo interativo do IronPython:

```
ipy
```

Para executar um programa:

```
ipy prog.py
```

As bibliotecas do CPython podem ser usadas dentro do IronPython, desde que as versões sejam compatíveis.

Exemplo:

```
import sys

sys.path.append(r'c:\python25\lib')
import os
print os.listdir('.')
```

Acrescenta no *PYTHONPATH*.

Exemplo usando um componente .NET:

```
from System.Diagnostics import Process
Process.Start('http://www.w3c.org/')
```

A função *Start* irá evocar o *browser* para abrir a URL.

Os objetos .NET podem ser usados ao invés dos *builtins* do Python:

```
import System
from System.Collections import  Hashtable

hash = Hashtable()
hash['baixo'] = '4 cordas'
hash['guitarra'] = '6 cordas'

for item in hash:
    print item.Key, '=>', item.Value
```

A classe *Hashtable* tem funcionalidade semelhante ao dicionário do Python.

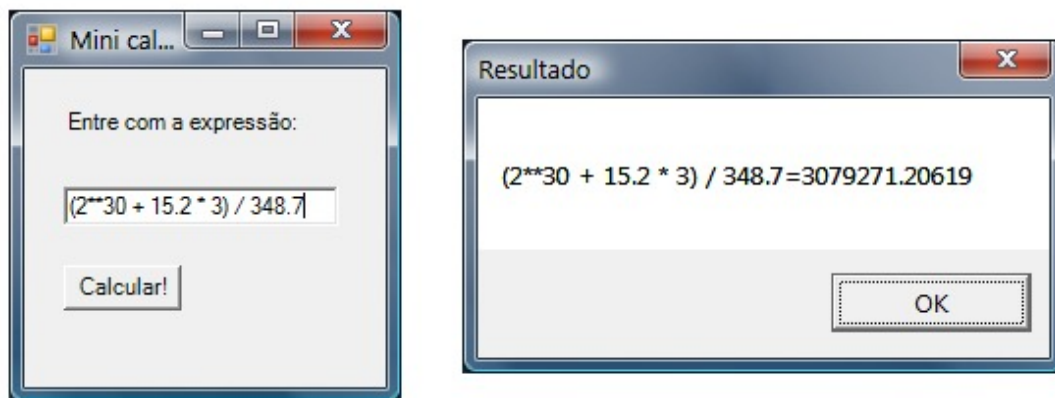
Integração com outros componentes .NET adicionais, como o *Windows Forms*, que

implementa a interface gráfica, é feita através do módulo clr. Após a importação do módulo, o IronPython passa a usar os tipos do .NET, ao invés da biblioteca padrão do Python.

Exemplo com *Windows Forms*:

```
# -*- coding: utf-8 -*-

import clr

# Adiciona referências para esses componentes
clr.AddReference('System.Windows.Forms')
clr.AddReference('System.Drawing')

# Importa os componentes
from System.Windows.Forms import *
from System.Drawing import *

# Cria uma janela
frm = Form(Width=200, Height=200)

# Coloca título na janela
frm.Text = 'Mini calculadora Python'

# Cria texto
lbl = Label(Text='Entre com a expressão:',
    Left=20, Top=20, Width=140)
# Adiciona a janela
frm.Controls.Add(lbl)

# Cria caixa de texto
txt = TextBox(Left=20, Top=60, Width=140)
# Adiciona a janela
frm.Controls.Add(txt)

# Função para o botão
def on_btn_click(*args):

    try:
        r = repr(eval(txt.Text))
        MessageBox.Show(txt.Text + '=' + r, 'Resultado')

    except:
        MessageBox.Show('Não foi possível avaliar: ' + \
            txt.Text, 'Erro')

# Cria botão
btn = Button(Text='Calcular!', Left=20, Top=100, Width=60)
btn.Click += on_btn_click
# Adiciona a janela
frm.Controls.Add(btn)
```

```
# Mostra a janela
frm.Show()

# Aplicação entra no loop de eventos,
# esperando pela interação do usuário
Application.Run(frm)
```

Interface do programa:

O mais comum é usar herança para especializar a janela, em uma solução mais orientada a objetos, encapsulando o código da criação e manipulação dos controles.A segunda versão do programa usa herança e inclui um componente de *layout*: *FlowLayoutPanel*.

```
# -*- coding: utf-8 -*-
"""
Mini calculadora Python
"""
import clr

clr.AddReference('System.Windows.Forms')
clr.AddReference('System.Drawing')

from System.Windows.Forms import *
from System.Drawing import *

class Janela(Form):
    """
```

```
    Janela principal
    """
    def __init__(self):
        """
        Inicializa a janela
        """

        self.Width=200
        self.Height=200

        self.Text = 'Mini calculadora Python'

        self.lbl = Label(Text='Entre com a expressão:')

        self.txt = TextBox()

        self.btn = Button(Text='Calcular!')
        self.btn.Click += self.on_btn_click

        # Layout automático para os controles
        self.panel = FlowLayoutPanel(Dock = DockStyle.Fill)
        self.panel.Controls.Add(self.lbl)
        self.panel.Controls.Add(self.txt)
        self.panel.Controls.Add(self.btn)
        self.Controls.Add(self.panel)

        self.Show()

        Application.Run(self)

    def on_btn_click(self, *args):
        """
        Acontece quando o botão é pressionado
        """

        try:
            r = repr(eval(self.txt.Text))
            MessageBox.Show(self.txt.Text + ' = ' + r, 'Resultado')

        except:
            MessageBox.Show('Não foi possível avaliar: ' + \
                self.txt.Text, 'Erro')

if __name__ == '__main__':

    janela = Janela()
```

O IronPython pode ser usado com o Mono[72], que uma implementação *Open Source* da especificação do .NET. O Mono apresenta a vantagem de ser portável, suportando outras plataformas além do Windows, porém não implementa todos os componentes do .NET

72 Endereço do projeto: http://www.mono-project.com/Main_Page.

(como o *Windows Forms*). Existe também uma IDE para o IronPython, chamada IronPython Studio[73].

73 Disponível em: http://www.codeplex.com/IronPythonStudio.

Respostas dos exercícios I

1. Implementar duas funções:
   - Uma que converta temperatura em graus *Celsius* para *Fahrenheit*.
   - Outra que converta temperatura em graus *Fahrenheit* para *Celsius*.

Lembrando que:

$$F = \frac{9}{5} \cdot C + 32$$

Solução:

```
def celsius_fahrenheit(c=0):

    # round(n, d) => arredonda n em d casas decimais
    return round(9. * c / 5. + 32., 2)

def fahrenheit_celsius(f=0):

    return round(5. * (f - 32.) / 9., 2)

# Testes
print celsius_fahrenheit(123.0)
print fahrenheit_celsius(253.4)
```

2. Implementar uma função que retorne verdadeiro se o número for primo (falso caso contrário). Testar de 1 a 100.

Solução:

```
# -*- coding: latin1 -*-

# Testa se o número é primo
def is_prime(n):

    if n < 2:
        return False

    for i in range(2, n):
        if not n % i:
            return False
    else:
        return True
```

```
# Para x de 1 a 100
for x in range(1, 101):
    if is_prime(x):
        print x
```

3. Implementar uma função que receba uma lista de listas de comprimentos quaisquer e retorne uma lista de uma dimensão.

Solução:

```
def flatten(it):
    """
    "Achata" listas...
    """

    # Se for uma lista
    if isinstance(it, list):
        ls = []

        # Para cada item da lista
        for item in it:
            # Evoca flatten() recursivamente
            ls = ls + flatten(item)
        return ls

    else:
        return [it]

# Teste
l = [[1, [2]], [3, 4], [[5, 6], 7]]
print flatten(l)

# imprime: [1, 2, 3, 4, 5, 6, 7]
```

4. Implementar uma função que receba um dicionário e retorne a soma, a média e a variação dos valores.

Solução:

```
# -*- coding: latin1 -*-

def stat(dic):

    # Soma
    s = sum(dic.values())
```

```
    # Média
    med = s / len(dic.values())

    # Variação
    var = max(dic.values()) - min(dic.values())

    return s, med, var
```

5. Escreva uma função que:
- Receba uma frase como parâmetro.
- Retorne uma nova frase com cada palavra com as letras invertidas.

Solução:

```
def reverse1(t):
    """
    Usando um loop convencional.
    """

    r = t.split()
    for i in xrange(len(r)):
        r[i] = r[i][::-1]
    return ' '.join(r)

def reverse2(t):
    """
    Usando Generator Expression.
    """

    return ' '.join(s[::-1] for s in t.split())

# Testes
f = 'The quick brown fox jumps over the lazy dog'
print reverse1(f)
print reverse2(f)
# mostra: "ehT kciuq nworb xof spmuj revo eht yzal god"
```

6. Crie uma função que:
- Receba uma lista de tuplas (dados), um inteiro (chave, zero por padrão igual) e um booleano (reverso, falso por padrão).
- Retorne dados ordenados pelo item indicado pela chave e em ordem decrescente se reverso for verdadeiro.

Solução:

```
def ord_tab(dados, chave=0, reverso=False):

    # Rotina para comparar as tuplas em sort()
    def _ord(x, y):

        return x[chave] - y[chave]

    dados.sort(_ord, reverse=reverso)

    return dados

# Testes
t = [(1, 2, 0), (3, 1, 5), (0, 3, 3)]
print ord_tab(t)
print ord_tab(t, 1)
print ord_tab(t, 2)

# Mostra:
# [(0, 3, 3), (1, 2, 0), (3, 1, 5)]
# [(3, 1, 5), (1, 2, 0), (0, 3, 3)]
# [(1, 2, 0), (0, 3, 3), (3, 1, 5)]
```

Respostas dos exercícios II

1. Implementar um programa que receba um nome de arquivo e gere estatísticas sobre o arquivo (número de caracteres, número de linhas e número de palavras)

Solução 1:

(Economizando memória)

```
# -*- coding: latin1 -*-

filename = raw_input('Nome do arquivo: ')
in_file = file(filename)

c, w, l = 0, 0, 0

# Para cada linha do arquivo
for line in in_file:

    # Soma 1 ao número de linhas
    l += 1

    # Soma o tamanho da linha ao número de caracteres
    c += len(line)

    # Soma a quantidade de palavra
    w += len(line.split())

in_file.close()

print 'Bytes: %d, palavras: %d, linhas: %s' % (c, w, l)
```

Solução 2:

(Economizando código)

```
# -*- coding: latin1 -*-

filename = raw_input('Nome do arquivo: ')

# Lê o arquivo inteiro para uma string
chars = file(filename).read()

c = len(chars)
w = len(chars.split())
```

```
# Soma o número de caracteres de nova linha
l = chars.count('\n')

print 'Bytes: %d, palavras: %d, linhas: %s' % (c, w, l)
```

2. Implementar um módulo com duas funções:

- *matrix_sum(*matrices)*, que retorna a matriz soma de matrizes de duas dimensões.
- *camel_case(s)*, que converte nomes para CamelCase.

Solução:

```
# -*- coding: latin1 -*-

def matrix_sum(*matrices):
    """
    Soma matrizes de duas dimensões.
    """
    # Pegue a primeira matriz
    mat = matrices[0]

    # Para cada matriz da segunda em diante
    for matrix in matrices[1:]:

        # Para cada linha da matriz
        for x, row in enumerate(matrix):

            # Para cada elemento da linha
            for y, col in enumerate(row):

                # Some na matriz de resposta
                mat[x][y] += col

    return mat

def camel_case(s):
    """
    Formata strings DestaForma.
    """
    return ''.join(s.title().split())

if __name__ == '__main__':

    # Testes
    print matrix_sum([[1, 2], [3, 4]], [[5, 6], [7, 8]])
    print camel_case('close to the edge')
```

3. Implementar uma função que leia um arquivo e retorne uma lista de tuplas com os dados (o separador de campo do arquivo é vírgula), eliminando as linhas vazias. Caso

ocorra algum problema, imprima uma mensagem de aviso e encerre o programa.

Script para gerar os dados de teste:

```
# -*- coding: latin1 -*-

# Importa o módulo para gerar
# números randômicos
import random

# Abre o arquivo
csv = file('test.csv', 'w')

for i in xrange(100):
    r = []

    for i in xrange(10):
        # random.randrange() escolhe números
        # dentro de um intervalo. A sintaxe
        # é a mesma da função range()
        r.append('%04d' %random.randrange(1000))

    csv.write(','.join(r) + '\n')

# Fecha o arquivo
csv.close()
```

Solução:

```
# -*- coding: latin1 -*-

def load_csv(fn):

    try:

        # Lê todas as linhas do arquivo
        lines = file(fn).readlines()
        new_lines = []

        for line in lines:
            new_line = line.strip()

            # Se houver caracteres na linha
            if new_line:

                # Quebra nas vírgulas, converte para tupla e
                # acrescenta na saída
                new_lines.append(tuple(new_line.split(',')))
```

```
        return new_lines

    # Tratamento de exceção
    except:

        print 'Ocorreu um erro ao ler o arquivo', fn
        raise SystemExit
```

4. Implementar um módulo com duas funções:
   - *split(fn, n)*, que quebra o arquivo fn em partes de n *bytes* e salva com nomes seqüenciais (se *fn* = arq.txt, então arq_001.txt, arq_002.txt, ...)
   - *join(fn, fnlist)* que junte os arquivos da lista *fnlist* em um arquivo só *fn*.

Solução:

```
# -*- coding: latin1 -*-
"""
breaker.py
"""

# Quebra o arquivo em fatias de n bytes
def split(fn, n):

    bytes = list(file(fn, 'rb').read())
    name, ext = fn.split('.')
    num = 1

    while bytes:
        out = ''.join(bytes[:n])
        del bytes[:n]
        newfn = '%s_%3d.%s' % (name, num, ext)
        file(newfn, 'wb').write(out)
        num += 1

# Junta as fatias em um arquivo
def join(fn, fnlist):

    out = ''
    for f in fnlist:
        out += file(f, 'rb').read()
    file(fn, 'wb').write(out)

if __name__ == '__main__':
    # Teste
    import glob

    split('breaker.py', 20)
    join('breaker2.py', sorted(glob.glob('breaker_*.py')))
```

5. Crie um *script* que:
- Compare a lista de arquivos em duas pastas distintas.
- Mostre os nomes dos arquivos que tem conteúdos diferentes e/ou que existem em apenas uma das pastas.

Solução:

```
# -*- coding: latin1 -*-

import os

# Nomes das pastas
pst1 = 'teste1'
pst2 = 'teste2'

# Lista o conteúdo das pastas
lst1 = os.listdir(pst1)
lst2 = os.listdir(pst2)

for fl in lst1:

    if fl in lst2:

        # Lê os arquivos e compara:
        if file(os.path.join(pst1, fl)).read() <> \
            file(os.path.join(pst2, fl)).read():
            print fl, 'diferente'

    # O arquivo não está na segunda pasta
    else:
        print fl, 'apenas em', pst1

for fl in lst2:
    # O arquivo não está na primeira pasta
    if not fl in lst1:
        print fl, 'apenas em', pst2
```

6. Faça um *script* que:
- Leia um arquivo texto.
- Conte as ocorrências de cada palavra.
- Mostre os resultados ordenados pelo número de ocorrências.

Solução:

```
# -*- coding: latin1 -*-
```

```
import string

# Lê o arquivo
texto = file('note.txt').read()
texto_limpo = ''

# Limpa o texto
for car in texto:
    if not car in string.punctuation:
        texto_limpo += car

# Separa as palavras
palavras = texto_limpo.split()

# Conta
resp = {}
for palavra in palavras:
    resp[palavra] = resp.get(palavra, 0) + 1
saida = resp.items()

# Ordena
def cmp(x, y):
    return x[-1] - y[-1]
saida.sort(cmp=cmp, reverse=True)

# Imprime
for k, v in saida:
    print k, '=>', v
```

Respostas dos exercícios III

1. Implementar um gerador de números primos.

Solução:

```
# -*- coding: latin1 -*-

# Verifica se o número é primo
def is_prime(n):

    if n < 2:
        return False

    for i in xrange(2, n):
        if not n % i:
            return False

    else:
        return True

# Gerador de números primos
def prime_gen():
    i = 1

    while True:
        if is_prime(i): yield i
        i += 1

# Teste: 100 primeiros primos
prime_iter = prime_gen()

for i in range(100):
    print prime_iter.next()
```

2. Implementar o gerador de números primos como uma expressão (dica: use o módulo *itertools*).

Solução:

```
# -*- coding: latin1 -*-

from itertools import count

# Verifica se o número é primo
def is_prime(n):
```

```
    if n < 2:
        return False

    for i in xrange(2, n):
        if not n % i:
            return False
    else:
        return True

# Generator Expression
primes = (i for i in count() if is_prime(i))

# Teste: 100 primeiros primos
for i in range(100):
    print primes.next()
```

3. Implementar um gerador que produza tuplas com as cores do padrão RGB (R, G e B variam de 0 a 255) usando *xrange()* e uma função que produza uma lista com as tuplas RGB usando *range()*. Compare a performance.

Solução:

```
# -*- coding: latin1 -*-

def rgb_lst():

    rgb = []
    for r in range(256):
        for g in range(256):
            for b in range(256):
                rgb.append((r, g, b))

    return rgb

def rgb_gen():

    for r in xrange(256):
        for g in xrange(256):
            for b in xrange(256):

                yield (r, g, b)

import time

tt = time.time()
l = rgb_lst()
print time.time() - tt

tt = time.time()
```

```
for color in rgb_gen(): pass
print time.time() - tt
```

4. Implementar um gerador que leia um arquivo e retorne uma lista de tuplas com os dados (o separador de campo do arquivo é vírgula), eliminando as linhas vazias. Caso ocorra algum problema, imprima uma mensagem de aviso e encerre o programa.

Solução:

```
# -*- coding: latin1 -*-

def load_csv(fn):

    try:
        for line in file(fn):
            new_line = line.strip()

            if new_line:
                yield tuple(new_line.split(','))

    except:
        print 'Ocorreu um erro ao ler o arquivo', fn
        raise SystemExit

# Teste
for line in load_csv('teste.csv'):
    print line
```

Respostas dos exercícios IV

1. Crie uma classe que modele um quadrado, com um atributo lado e os métodos: mudar valor do lado, retornar valor do lado e calcular área.

Solução:

```
# -*- coding: latin1 -*-

class Square(object):
    """
    Classe que modela um quadrado.
    """

    def __init__(self, side=1):

        self.side = side

    def get_side(self):

        return self.side

    def set_side(self, side):

        self.side = side

    def get_area(self):

        # A área é o quadrado do lado
        return self.side ** 2

# Testes
square = Square(2)
square.set_side(3)
print square.get_area()
```

2. Crie uma classe derivada de lista com um método retorne os elementos da lista sem repetição.

Solução:

```
# -*- coding: latin1 -*-

class List(list):

    def unique(self):
        """
```

```
        Retorna a lista sem repetições.
        """

        res = []
        for item in self:

            if item not in res:
                res.append(item)

        return res

# Teste
l = List([1, 1, 2, 2, 2, 3, 3])

print l.unique()
```

3. Implemente uma classe *Carro* com as seguintes propriedades:
   - Um veículo tem um certo consumo de combustível (medidos em km / litro) e uma certa quantidade de combustível no tanque.
   - O consumo é especificado no construtor e o nível de combustível inicial é 0.
   - Forneça um método *mover(km)* que receba a distância em quilômetros e reduza o nível de combustível no tanque de gasolina.
   - Forneça um método *gasolina()*, que retorna o nível atual de combustível.
   - Forneça um método *abastecer(litros)*, para abastecer o tanque.

Solução:

```
# -*- coding: latin1 -*-

class Carro(object):
    """
    Classe que calcula o consumo de um carro.
    """

    tanque = 0

    def __init__(self, consumo):

        self.consumo = consumo

    def mover(self, km):

        gasto = self.consumo * km

        if self.tanque > gasto:
            self.tanque = self.tanque - gasto
        else:
```

```
        self.tanque = 0

    def abastecer(self, litros):

        self.tanque = self.tanque + litros

    def gasolina(self):

        return self.tanque

# Teste
carro = Carro(consumo=5)
carro.abastecer(litros=220)
carro.mover(km=20)
print carro.gasolina()
```

4. Implementar uma classe *Vetor*:

- Com coordenadas x, y e z.
- Que suporte soma, subtração, produto escalar, produto vetorial.
- Que calcule o módulo (valor absoluto) do vetor.

Solução:

```
# -*- coding: latin1 -*-

import math

class Vetor(object):

    def __init__(self, x, y, z):

        self.x = float(x)
        self.y = float(y)
        self.z = float(z)

    def __repr__(self):

        return 'Vetor(x=%.1f, y=%.1f, z=%.1f)' % (self.x, self.y, self.z)

    def __add__(self, v):

        x = self.x + v.x
        y = self.y + v.y
        z = self.z + v.z
        return Vetor(x, y, z)

    def __sub__(self, v):
```

```
        x = self.x - v.x
        y = self.y - v.y
        z = self.z - v.z
        return Vetor(x, y, z)

    def __abs__(self):

        tmp = self.x ** 2 + self.y ** 2 + self.z ** 2
        return math.sqrt(tmp)

    def __mul__(self, v):

        if isinstance(v, Vetor):
            x = self.y * v.z - v.y * self.z
            y = self.z * v.x - v.z * self.x
            z = self.x * v.y - v.x * self.y
        else:
            x = self.x * float(v)
            y = self.y * float(v)
            z = self.z * float(v)
        return Vetor(x, y, z)

vetor = Vetor(1, 2, 3)

print abs(vetor)
print Vetor(4.5, 5, 6) + vetor
print Vetor(4.5, 5, 6) - vetor
print Vetor(4.5, 5, 6) * vetor
print Vetor(4.5, 5, 6) * 5
```

5. Implemente um módulo com:

- Uma classe *Ponto*, com coordenadas x, y e z.
- Uma classe *Linha*, com dois pontos A e B, e que calcule o comprimento da linha.
- Uma classe *Triangulo*, com dois pontos A, B e C, que calcule o comprimento dos lados e a área.

Solução:

```
class Ponto(object):

    def __init__(self, x, y, z):

        # Coordenadas
        self.x = float(x)
        self.y = float(y)
        self.z = float(z)

    def __repr__(self):
```

```
        return '(%2.1f, %2.1f, %2.1f)' % \
        (self.x, self.y, self.z)

class Linha(object):

    def __init__(self, a, b):

        # Pontos
        self.a = a
        self.b = b

    def comp(self):

        x = self.b.x - self.a.x
        y = self.b.y - self.a.y
        z = self.b.z - self.a.z

        return round((x ** 2 + y ** 2 + z ** 2)\
            ** .5, 1)

    def __repr__(self):

        return '%s => %s' % \
        (self.a, self.b)

class Triangulo(object):

    def __init__(self, a, b, c):

        # Vertices
        self.a = a
        self.b = b
        self.c = c

        # Lados
        self.ab = Linha(a, b)
        self.bc = Linha(b, c)
        self.ca = Linha(c, a)

    def area(self):

        # Comprimento dos lados
        ab = self.ab.comp()
        bc = self.bc.comp()
        ca = self.ca.comp()

        # Semiperimetro
        p = (ab + bc + ca) / 2.

        # Teorema de Heron
        return round((p * (p - ab) * (p - bc) \
```

```
        * (p - ca)) ** .5, 1)

    def __repr__(self):

        return '%s => %s => %s)' % \
        (self.a, self.b, self.c)

# Testes
a = Ponto(2, 3, 1)
b = Ponto(5, 1, 4)
c = Ponto(4, 2, 5)
l = Linha(a, b)
t = Triangulo(a, b, c)

print 'Ponto A:', a
print 'Ponto B:', b
print 'Ponto C:', c
print 'Linha:', l
print 'Comprimento:', l.comp()
print 'Triangulo:', t
print 'Area:', t.area()

# Mostra:
# Ponto A: (2.0, 3.0, 1.0)
# Ponto B: (5.0, 1.0, 4.0)
# Ponto C: (4.0, 2.0, 5.0)
# Linha: (2.0, 3.0, 1.0) => (5.0, 1.0, 4.0)
# Comprimento: 4.7
# Triangulo: (2.0, 3.0, 1.0) => (5.0, 1.0, 4.0) => (4.0, 2.0, 5.0))
# Area: 3.9
```

Respostas dos exercícios V

1. Implementar uma classe *Animal* com os atributos: nome, espécie, gênero, peso, altura e idade. O objeto derivado desta classe deverá salvar seu estado em arquivo com um método chamado "salvar" e recarregar o estado em um método chamado "desfazer".

Solução:

```
# -*- coding: latin1 -*-

import pickle

class Animal(object):
    """
    Classe que representa um animal.
    """

    attrs = ['nome', 'especie', 'genero', 'peso', 'altura', 'idade']

    def __init__(self, **args):

        # Crie os atributos no objeto a partir da lista
        # Os atributos tem None como valor default
        for attr in self.attrs:
            setattr(self, attr, args.get(attr, None))

    def __repr__(self):

        dic_attrs = {}
        for attr in self.attrs:
            dic_attrs[attr] = getattr(self, attr)
        return 'Animal: %s' % str(dic_attrs)

    def salvar(self):

        """
        Salva os dados do animal.
        """
        dic_attrs = {}
        for attr in self.attrs:
            dic_attrs[attr] = getattr(self, attr)

        pickle.dump(dic_attrs, file('a.pkl', 'w'))

    def desfazer(self):

        """
        Restaura os últimos dados salvos.
        """
```

```
        attrs = pickle.load(file('a.pkl'))

        for attr in attrs:
            setattr(self, attr, attrs[attr])

# Teste
gato = Animal(nome='Tinker', especie='Gato', genero='m',
    peso=6, altura=0.30, idade=4)

gato.salvar()
gato.idade = 5
print gato
gato.desfazer()
print gato
```

2. Implementar uma função que formate uma lista de tuplas como tabela HTML.

Solução:

```
# -*- coding: latin1 -*-

# O módulo StringIO implementa uma classe
# de strings que se comportam como arquivos
import StringIO

def table_format(dataset):
    """
    Classe que representa um animal.
    """

    out = StringIO.StringIO()
    out.write('<table>')

    for row in dataset:
        out.write('<tr>')
        for col in row:
            out.write('<td>%s</td>' % col)
        out.write('</tr>')

    out.write('</table>')
    out.seek(0)
    return out.read()
```

3. Implementar uma aplicação *Web* com uma saudação dependente do horário (exemplos: “Bom dia, são 09:00.”, “Boa tarde, são 13:00.” e “Boa noite, são 23:00.”).

Solução:

```
# -*- coding: latin1 -*-

import time
import cherrypy

class Root(object):
    """
    Raiz do site.
    """

    @cherrypy.expose
    def index(self):
        """
        Exibe a saudação conforme o horário do sistema.
        """

        # Lê a hora do sistema
        hour = '%02d:%02d' % time.localtime()[3:5]

        if '06:00' < hour <= '12:00':
            salute = 'Bom dia'
        elif '12:00' < hour <= '18:00':
            salute = 'Boa tarde'
        else:
            salute = 'Boa noite'

        # Retorna a mensagem para o browser
        return '%s, são %s.' % (salute, hour)

cherrypy.quickstart(Root())
```

4. Implementar uma aplicação *Web* com um formulário que receba expressões Python e retorne a expressão com seu resultado.

Solução:

```
# -*- coding: latin1 -*-

import traceback
import cherrypy

class Root(object):

    # Modelo para a página HTML
    template = '''
    <html><body>
    <form action="/">
    <input type="text" name="exp" value="%s" />
    <input type="submit" value="enviar">
```

```
    <pre>%s</pre>
    </body></html>'''

    @cherrypy.expose
    def index(self, exp=''):

        out = ''
        if exp:

            # Tente avaliar a expressão
            try:
                out = eval(exp)

            # Se der errado, mostre a mensagem do erro
            except:
                out = traceback.format_exc()

        return self.template % (exp, out)

cherrypy.quickstart(Root())
```

Respostas dos exercícios VI

1. Implementar um módulo com uma função *tribonacci(n)* que retorne uma lista de n números de Tribonacci, aonde n é o parâmetro da função. Faça testes da função caso o módulo seja executado como principal.

Solução:

```
# -*- coding: latin1 -*-

def tribonacci(n):
    """
    Retorna uma lista com n elementos de Tribonacci.

    >>> t = [1, 1, 2, 4, 7, 13, 24, 44, 81, 149, \
    274, 504, 927, 1705, 3136, 5768, 10609, 19513, \
    35890, 66012, 121415, 223317]
    >>> t == tribonacci(22)
    True
    >>> tribonacci('22')
    Traceback (most recent call last):
      File "pyro_server.py", line 26, in <module>
        print Dist().tribonacci('22')
      File "pyro_server.py", line 14, in tribonacci
        raise TypeError
    TypeError
    """
    if type(n) is not int:
        raise TypeError

    # Os 3 primeiros elementos da seqüência
    t = [1, 1, 2]

    if n < 4:
        return t[:n]

    for i in range(3, n):

        # Soma os 3 elementos finais
        t.append(sum(t[-3:]))

    return t

def _doctest():
    """
    Evoca o doctest.
    """

    import doctest
```

```
    doctest.testmod()

if __name__ == "__main__":

    _doctest()
```

2. Implementar:
- um servidor que publique um objeto distribuído e este evoque a função *tribonacci*.
- um cliente que use o objeto distribuído para calcular a seqüência de Tribonacci.

Solução:

Servidor:

```
# -*- coding: latin1 -*-

import Pyro.core

# Importa o módulo com a função
import trib

class Dist(Pyro.core.ObjBase):

    @staticmethod
    def tribonacci(n):
        return trib.tribonacci(n)

if __name__ == '__main__':

    # Define a porta TCP/IP usada pelo Pyro
    Pyro.config.PYRO_PORT = 8888

    # Define o limite de cliente ao mesmo tempo
    Pyro.config.PYRO_MAXCONNECTIONS = 2000

    Pyro.core.initServer()

    # norange=1 faz com que o Pyro sempre use a mesma porta
    daemon = Pyro.core.Daemon(norange = 1)

    # Define o limite de tempo
    daemon.setTimeout(300)

    uri = daemon.connect(Dist(),'dist')
    daemon.requestLoop()
```

Cliente:

```
# -*- coding: latin1 -*-

import Pyro.core

# URL com a porta
url = 'PYROLOC://127.0.0.1:8888/dist'
proxy = Pyro.core.getProxyForURI(url)

# Teste com até dez elementos
for i in range(10):
   print i + 1, '=>', proxy.tribonacci(i + 1)
```

Saída:

```
Pyro Client Initialized. Using Pyro V3.7
1 => [1]
2 => [1, 1]
3 => [1, 1, 2]
4 => [1, 1, 2, 4]
5 => [1, 1, 2, 4, 7]
6 => [1, 1, 2, 4, 7, 13]
7 => [1, 1, 2, 4, 7, 13, 24]
8 => [1, 1, 2, 4, 7, 13, 24, 44]
9 => [1, 1, 2, 4, 7, 13, 24, 44, 81]
10 => [1, 1, 2, 4, 7, 13, 24, 44, 81, 149]
```

Índice remissivo